PYTHON 3.14 · С НУЛЯ

Python с нуля

программирование, графика, приложения и игры

Siergej Sobolewski

Software & AI Engineer, основатель Cartesian School

Об авторе

Сергей Соболевски (Siergej Sobolewski) — Software & AI Engineer, основатель Cartesian School. Эта книга выросла из практического вопроса: как объяснить программирование человеку, который никогда раньше не писал ни строчки кода, не превращая объяснение ни в скучный справочник, ни в поверхностную «книжку с картинками».

Cartesian School — образовательный проект, в основе которого лежит идея, что современный инженерный подход и понятное объяснение для начинающих не противоречат друг другу: одно и то же занятие может быть одновременно точным и увлекательным.

О техническом рецензенте

Открытый пункт

Имя технического рецензента будет добавлено сюда после того, как Product Owner назначит рецензента издания. Раздел сохранён по канонической структуре оглавления и не удалён — при этом мы не стали придумывать несуществующего человека.

Каждая глава этой книги проходит техническую проверку: весь код автоматически компилируется, проверяется линтером `ruff`, а примеры

в формате Jupyter Notebook реально выполняются от начала до конца — результаты вычислений в тексте книги получены не «на глаз», а настоящим запуском кода на Python 3.14.

Введение

Коротко о том, как устроена эта книга и как получить от неё максимум.

Для кого эта книга

Эта книга — для тех, кто раньше не писал код. Совсем. Неважно, сколько вам лет: десять, двадцать пять или пятьдесят — если вы умеете пользоваться компьютером и готовы разбираться, книга проведёт вас от первой напечатанной строки до собственных игр, приложений и рисунков на экране.

Как устроена каждая глава

Почти в каждом разделе вы встретите один и тот же порядок: сначала — зачем это нужно, затем — простое объяснение идеи, потом — рабочий пример, который можно запустить прямо сейчас, и в конце — практика и типичные ошибки. Мы намеренно не начинаем с сухого определения — сначала должно быть понятно, зачем оно вам, а термин появится следом.

Классический подход и современный Python 3.14

Python существует больше тридцати лет, и за это время в языке появлялись более удобные способы делать привычные

вещи. Там, где это важно, книга сначала показывает классический приём — потому что именно его вы встретите в старом коде, статьях и ответах на форумах — а затем современный вариант для Python 3.14 и объясняет, чем они отличаются и что использовать сегодня. Это отмечено специальным блоком «Классический подход → современный Python».

Теория на сайте, практика в Jupyter Notebook

Теорию вы читаете здесь, на сайте Cartesian School — с подсветкой кода, поиском и навигацией. А для практики к каждому разделу прилагается

отдельный файл Jupyter Notebook: в нём можно менять код и сразу видеть результат, не открывая ничего дополнительного, кроме Visual Studio Code или PyCharm.

Уровни сложности практики

Задания отмечены звёздами: ★ — базовая практика, повторяющая пример из раздела; ★★ — самостоятельная задача, требующая немного подумать; ★★★ — задача повышенной сложности для тех, кто хочет большего. Не в каждом разделе есть все три уровня — только там, где это оправдано.

Что понадобится

Python 3.14, установленный по инструкции из главы 2, и один из двух редакторов кода: Visual Studio Code или PyCharm — оба подробно описаны в книге. Больше почти ничего не нужно: все проекты в этой книге можно запустить на обычном ноутбуке.

ГЛАВА 1 · СТР. 1

А вы знали?

Первая глава книги «Python с нуля» — что такое программирование, зачем оно нужно и почему Python отлично подходит для первого языка.

Что такое программирование?

Прежде чем писать код, полезно понять, что вообще происходит, когда компьютер «выполняет программу» — и почему для первого языка мы выбрали именно Python.

Что такое программирование?

Компьютер невероятно быстрый, но абсолютно несамостоятельный. Сам по себе он не умеет вообще ничего — ни открыть игру, ни показать фотографию, ни посчитать сдачу в магазине. Всё, что делает компьютер, он делает потому, что кто-то заранее написал для него точную последовательность команд. Эта последовательность и называется **программой**, а процесс её написания — **программированием**.

Текст программы, который пишет человек, называется **исходным кодом** (source code). Компьютер не понимает исходный код напрямую — между вашим

текстом и процессором стоит специальная программа-переводчик. Для Python она называется **интерпретатором**: он построчно читает ваш код и превращает его в действия, которые понимает компьютер, — примерно как синхронный переводчик на переговорах.

Терминал

Программу, где вы вводите текстовые команды и видите текстовый ответ компьютера, называют **терминалом** (или консолью, командной строкой). Именно там запускают программы на Python — подробно об этом в главе 3.

Хорошая новость в том, что программирование — это не «разговор с компьютером на особом тайном языке», а прежде всего умение **точно объяснить порядок действий**. Если вы хоть раз объясняли кому-то дорогу или писали рецепт по шагам — вы уже тренировали именно тот навык, на котором строится программирование.

Почему вашим детям стоит научиться программировать?

Программирование — не заявка на профессию «программист», а способ мышления, который пригождается в любой области. Это касается и взрослых,

которые садятся за первую программу сами. Три причины, почему этому стоит учиться в любом возрасте:

- **Ошибка — это нормально.** Программа почти никогда не работает с первого раза, и это не повод расстраиваться — это часть процесса. Программирование учит спокойно находить причину и пробовать снова, вместо того чтобы бояться ошибиться.
- **Идея превращается в результат.** Мысль «а что если сделать вот так» можно проверить за несколько минут, написав и запустив код, — не нужно ничьё разрешение.
- **Это переносимый навык.** Умение разложить сложную задачу на

простые шаги пригождается далеко за пределами программирования — в учёбе, работе и повседневных задачах.

Почему Python?

Языков программирования — сотни. Python выбран для первого языка по трём практическим причинам:

- **Читается почти как обычный текст.** Строка `if age >= 18:` понятна без перевода. Меньше отвлечённого синтаксиса — больше внимания на саму идею.
- **Один язык — очень разные результаты.** На Python пишут игры, рисуют графику, собирают

настольные и веб-приложения, обучают модели искусственного интеллекта. Всё, о чём вы прочитаете в этой книге, — на одном и том же языке.

- **Огромное сообщество.** Если у вас возникнет вопрос — скорее всего, кто-то уже задавал его и получил ответ. Готовых инструментов (их называют библиотеками или пакетами) для Python написано огромное количество, и в этой книге вы начнёте пользоваться некоторыми из них — например, `turtle` для графики.

Python — это весело!

Короткая экскурсия по тому, что вы будете строить своими руками на страницах этой книги — задолго до того, как станете экспертом.

Игры!

В главах 17, 19 и 21 вы соберёте три настоящие игры целиком: «Крестики-нолики», «Змейку» и космический шутер — от пустого окна до готовой игры с очками и правилами. Каждая из них

строится маленькими понятными шагами, а не появляется одним большим куском кода.

Графика и анимация

Уже в главе 6 вы начнёте рисовать на экране с помощью модуля `turtle` — виртуальной черепашки с пером, которая двигается по вашим командам. Вот как выглядит код, рисующий квадрат — просто для примера, разбирать его подробно будем позже:

podglyadyvaem_vpered.py

```
import turtle

artist = turtle.Turtle()
for _ in range(4):
    artist.forward(100)   # едем вперёд
    artist.right(90)      # поворачиваем
на 90°
```

Забегая вперёд

Не пытайтесь понять этот код прямо сейчас — `for` (цикл) появится только в главе 10. Здесь достаточно увидеть, что настоящий рисунок получается всего из нескольких строк.

Веб-сайты

В главе 22 вы узнаете, как устроен веб — что делают HTML, CSS и JavaScript в браузере, какую роль играет Python на сервере, — и запустите собственный работающий сайт на Flask.

Приложения

С помощью модуля `Tkinter` (главы 16–18) вы соберёте настоящие оконные приложения с кнопками и полями ввода — калькулятор чаевых и собственное приложение для рисования, — а не только программы, которые выводят текст в терминал.

Как получить максимум от этой книги

Несколько привычек, которые сделают чтение куда эффективнее, — и краткие итоги главы 1.

Четыре простые привычки сделают чтение книги гораздо эффективнее:

- **Печатайте код руками**, а не копируйте. Опечатка, которую вы находите и исправляете сами, учит куда лучше, чем код, который просто сработал с первого раза.

- **Меняйте числа и слова в примерах** — что произойдёт, если поставить не 100, а 300? Предсказание результата *до* запуска — лучшая тренировка.
- **Не пропускайте раздел «Типичная ошибка».** Там разобраны настоящие ошибки начинающих — увидеть их заранее полезнее, чем встретить их вслепую.
- **Открывайте ноутбук практики** к каждому разделу — теория на сайте объясняет идею, а ноутбук — то место, где вы её проверяете руками.

Если что-то не работает

Это не признак того, что программирование не для вас — это самая обычная часть процесса. В главе 3 и далее в книге разобрана целая система: как читать сообщение об ошибке и находить причину, а не бояться её.

Итоги

Что мы узнали в этой главе

- Программа — это точная последовательность команд для компьютера; исходный код превращает в действия **интерпретатор**.
- Программирование — переносимый навык: он учит раскладывать сложную задачу на простые шаги и спокойно относиться к ошибкам.
- Python выбран как первый язык за читаемость, универсальность применения и огромное сообщество.
- На страницах книги вы соберёте графику, игры,

оконные приложения и веб-сайт — всё на одном языке.

ГЛАВА 2 · СТР. 13

Давайте установим Python!

Устанавливаем настоящий Python 3.14 на Windows или Mac — шаг за шагом, с проверкой в терминале.

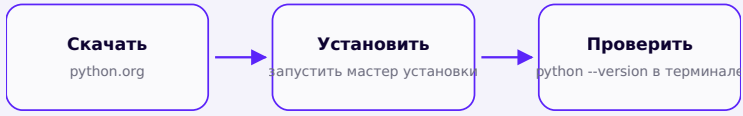
Говорим на языке компьютера

Прежде чем писать код, компьютеру нужно «выучить» Python — то есть получить интерпретатор, который умеет его понимать.

Начало работы — устанавливаем Python

Когда вы устанавливаете Python, вы устанавливаете **интерпретатор** — программу, которая умеет читать и выполнять код на Python (напомним: об этом шла речь в главе 1). После

установки ваш компьютер начинает «понимать» файлы с расширением `.py` и команду `python` в терминале.



Общая схема установки — детали различаются для Windows и Mac

Шаги отличаются в зависимости от операционной системы — выберите свою:

- Установка на Windows →
- Установка на Mac →

Скачивайте только с python.org

Официальный установщик — единственный источник, которому стоит доверять. Сайты-двойники и «ускоренные» установщики с других сайтов иногда добавляют в систему нежелательные программы. В этой книге мы всегда используем только python.org/downloads.

Зачем именно 3.14?

Python постоянно развивается. Версия 3.14 — самая новая стабильная версия на момент написания этой книги, и весь код в ней рассчитан именно на неё. Более старая версия (например, 3.9 или 3.10) в большинстве случаев тоже подойдёт для базовых глав, но несколько современных примеров в книге её потребуют.

Установка Python на компьютере с Windows

Три коротких шага: скачать официальный установщик, поставить галочку PATH и проверить результат в командной строке.

Скачивание Python

1. Откройте в браузере python.org/downloads/windows.
2. Найдите самую свежую версию 3.14.x и нажмите на ссылку **«Windows installer (64-bit)»**. Почти все современные компьютеры на

Windows — 64-битные; если сомневаетесь, эта версия почти наверняка подойдёт.

3. Дождитесь, пока файл вида `python-3.14.x-amd64.exe` загрузится в папку «Загрузки».

Установка Python

1. Запустите скачанный файл двойным щелчком.
2. На первом экране мастера установки обязательно поставьте галочку **«Add python.exe to PATH»** внизу окна. Это самый важный шаг: PATH — список папок, где Windows ищет программы по имени. Без этой

галочки команда `python` в терминале не будет найдена.

3. Нажмите **«Install Now»** и дождитесь завершения установки.
4. На последнем экране нажмите **«Close»**.

> **Забыли галочку PATH?**
>
> Не страшно — запустите установщик заново, выберите «Modify», затем на шаге «Advanced Options» включите «Add Python to environment variables» и завершите установку повторно.

Проверка установки

Откройте приложение **«Командная строка»** (наберите *cmd* в поиске Windows) и введите:

```
командная строка
```

```
python --version
```

Если вы видите строку вида `Python 3.14.x` — Python установлен и готов к работе. Переходите к главе 3.

Установка Python на устройстве Mac

Тот же принцип, что и на Windows, но с учётом особенностей macOS — и краткие итоги главы.

Скачивание Python

1. Откройте в браузере python.org/downloads/macos.
2. Найдите самую свежую версию 3.14.x и нажмите на ссылку **«macOS 64-bit universal2 installer»** — этот установщик подходит и для Mac с процессором Apple Silicon (M1/M2/M3/M4), и для Mac с процессором Intel.
3. Дождитесь, пока файл вида `python-3.14.x-macos11.pkg` загрузится в папку «Загрузки».

Установка Python

1. Откройте скачанный файл `.pkg` двойным щелчком.

2. Пройдите шаги мастера установки — «Введение», «Лицензия», «Место установки» — нажимая «Продолжить». Настройки по умолчанию подходят для этой книги, менять их не нужно.
3. На шаге «Тип установки» нажмите «Установить» и введите пароль своей учётной записи Mac, когда система попросит подтверждение.
4. Дождитесь надписи «Установка прошла успешно» и закройте окно.

На Mac уже есть Python — но не тот

На некоторых Mac предустановлена очень старая версия Python 2 для внутренних нужд системы. Не используйте её — после установки с python.org в системе появится отдельная, современная команда `python3`, именно её мы и будем использовать.

Проверка установки

Откройте приложение **«Терминал»** (Programs → Utilities → Terminal, либо через поиск Spotlight) и введите:

терминал

```
python3 --version
```

Если вы видите строку вида `Python 3.14.x` — всё готово.

Итоги

Что мы узнали в этой главе

- Установка Python — это установка **интерпретатора**, который умеет выполнять код на Python.
- Официальный и безопасный источник — только **python.org**.
- На Windows критически важно включить галочку **«Add python.exe to PATH»**.
- На Mac команда для запуска — **python3**, а не `python`.
- Проверить установку можно одной командой: `python --version` (Windows) или `python3 --version` (Mac).

ГЛАВА 3 · СТР. 27

Ваша первая программа на Python

Пишем и запускаем первую программу — в VS Code, в PyCharm и в интерактивной оболочке.

Создание и запуск программ на Python

Программа на Python начинается с обычного текстового файла. Разберём, как создать его и запустить — в VS Code и в PyCharm.

Программа на Python — это обычный текстовый файл с расширением `.py`. Чтобы его написать и запустить, нужен редактор кода. В этой книге мы используем два: **Visual Studio Code** и **PyCharm** — оба бесплатны (PyCharm — в редакции Community) и оба отлично подходят для Python.

Вариант A: Visual Studio Code

1. Установите VS Code с code.visualstudio.com и расширение **Python** (от Microsoft) через панель Extensions.

2. Создайте папку для проекта книги, например `python-s-nulya`, и откройте её через **File → Open Folder**.
3. Создайте новый файл `privet.py`.
4. В правом нижнем углу окна нажмите на индикатор версии Python и выберите установленный Python 3.14 в качестве интерпретатора — если индикатора нет, откройте палитру команд (`Ctrl+Shift+P` / `Cmd+Shift+P`) и выполните «Python: Select Interpreter».
5. Напишите код (пример — ниже) и нажмите на треугольник ▷ «Run Python File» в правом верхнем углу, либо откройте встроенный терминал

(`Ctrl+``) и наберите `python privet.py`.

Вариант B: PyCharm

1. Установите PyCharm Community с jetbrains.com/pycharm/download.
2. При создании проекта (**New Project**) укажите папку проекта и Python 3.14 в качестве интерпретатора — PyCharm покажет список найденных версий автоматически.
3. Щёлкните правой кнопкой по папке проекта в панели слева → **New → Python File**, назовите его `privet`.
4. Напишите код и нажмите на зелёный треугольник ▷ рядом со строкой `if __name__ ==` или в

верхней панели — либо кликните правой кнопкой в редакторе и выберите «Run».

Ваша первая программа

privet.py

```
print("Привет, Python!")
```

После запуска — любым из двух способов — вы должны увидеть в терминале:

вывод программы

```
Привет, Python!
```

Терминал внутри редактора

И VS Code, и PyCharm показывают вывод программы во встроенном терминале внизу окна — не нужно переключаться в отдельное приложение. Именно этим встроенным терминалом мы будем пользоваться на протяжении всей книги.

Интерактивный режим Python (Python Shell)

Второй способ работать с Python — вводить код по одной строке и сразу видеть результат, без файла и без print().

Интерактивный режим Python (Python Shell)

Кроме запуска файлов, у Python есть второй, не менее полезный режим — **интерактивная оболочка** (Python Shell, или REPL — Read-Eval-Print Loop: «прочитать — выполнить — напечатать — повторить»). В ней вы вводите код по одной строке и сразу видите результат — без файла и без «запуска» в привычном смысле.

Открыть её можно прямо в терминале VS Code или PyCharm, набрав `python` (Windows) или `python3` (Mac). Появится приглашение `>>>` — это значит, что оболочка ждёт вашу команду.

терминал

```
$ python
Python 3.14.0 (main, ...)
>>> print("Привет!")
Привет!
>>>
```

Как выйти

Наберите `exit()` и нажмите Enter, либо нажмите `Ctrl+Z` затем Enter (Windows) или `Ctrl+D` (Mac/Linux).

Ваша оболочка умеет считать

Раз оболочка сразу показывает результат каждой строки, её удобно использовать как быстрый калькулятор — без единого `print()`:

```
Python Shell
```

```
>>> 2 + 2
4
>>> 10 / 4
2.5
>>> 3 * 7
21
```

Это работает только в самой оболочке: в обычном `.py`-файле строка `2 + 2` сама по себе ничего не выведет — файл выполняется целиком и молча, если явно не попросить вывод через `print()`. Подробно об этом — в следующем разделе.

Вывод данных с помощью Python

print() умеет больше, чем кажется на первый взгляд — несколько значений сразу, свои разделители и управление концом строки.

В файле `.py` ничего не выводится на экран само по себе — нужно явно попросить об этом командой `print()`. Мы уже пользовались ей в главе 1, теперь разберём подробнее.

Несколько значений в одной команде

print_demo.py

```
print("Возраст:", 10, "лет")
```

Через запятую можно передать сколько угодно значений — `print()` сам вставит между ними пробел и выведет всё в одну строку.

Параметр `sep` — разделитель

По умолчанию значения разделяются пробелом. Это можно изменить параметром `sep`:

print_sep.py

```
print("2024", "01", "15", sep="-")
```

вывод программы

```
2024-01-15
```

Параметр `end` — чем закончить строку

По умолчанию каждый `print()` завершается переводом строки — поэтому следующий вызов начинается с новой строки. Параметр `end` позволяет это изменить:

print_end.py

```
print("Загрузка", end="")
print("...", end="")
print("готово!")
```

вывод программы

```
Загрузка...готово!
```

Пустая строка

Вызов `print()` без аргументов просто выводит пустую строку — удобно, чтобы отделить блоки текста друг от друга.

Режим сценариев IDLE

Python сам приносит с собой простой редактор — IDLE. Разберёмся, как им пользоваться и когда стоит перейти на более мощный инструмент.

У Python есть встроенный простой редактор кода, который устанавливается автоматически вместе с самим Python, — **IDLE** (Integrated Development and Learning Environment). Открыть его можно, найдя «IDLE» в меню Пуск (Windows) или Launchpad (Mac).

Режим сценариев (Script Mode)

При запуске IDLE открывается интерактивная оболочка — та же самая, что мы видели в предыдущем разделе, только в собственном окне. Чтобы написать полноценную программу, нужен отдельный файл: **File → New File** откроет пустое окно редактора — это и есть «режим сценариев».

1. Наберите код в новом окне редактора.
2. Сохраните файл: **File → Save** (расширение `.py` подставится само).
3. Запустите: **Run → Run Module**, либо просто нажмите клавишу `F5`.

4. Результат появится в окне интерактивной оболочки IDLE.

IDLE → современная разработка

IDLE (входит в Python «из коробки»)

```
print("Привет из IDLE!")
# запуск: меню Run -> Run Module (F5)
# один файл, минимум настроек
```

VS Code / PyCharm

```
print("Привет из VS Code!")
# запуск: кнопка Run или встроенный терминал
# подсветка ошибок на лету, автодополнение,
# отладчик, git, множество расширений
```

Что использовать сегодня: IDLE отлично подходит, чтобы буквально за одну минуту после установки Python написать и запустить первую строку кода — устанавливать ничего дополнительно не нужно. Но как только проекты вырастают за пределы одного файла, заметно не хватает автодополнения, отладчика и подсветки ошибок на лету. Начиная со следующей главы мы будем пользоваться VS Code или PyCharm — но IDLE стоит знать: именно её вы увидите, если попробуете Python на чужом компьютере без дополнительной настройки.

Практика: выведите своё имя (и кое-что ещё)

Собираем всё вместе: файл, терминал, print и выбор редактора — в одном небольшом упражнении.

Пора применить все четыре инструмента раздела на практике: файл, оболочку, print и IDLE (или VS Code/PyCharm — на ваш выбор) — в одном небольшом упражнении.

★ Базовая практика

Выведите своё имя

Создайте файл `o_sebe.py` и одной командой `print()` выведите своё имя.

★★ Самостоятельная задача

И кое-что ещё

В том же файле добавьте ещё две строки: одну — с вашим городом, вторую — с любым числом, использовав `sep`, чтобы вывести имя, город и число в одну строку через « · ».

★★★ Задача повышенной сложности

Тот же результат тремя способами

Получите один и тот же вывод тремя разными способами: запустив файл из терминала (`python o_sebe.py`), через кнопку Run в VS Code/ PyCharm, и через Run Module в IDLE.

Итоги

Что мы узнали в этой главе

- Программа на Python — обычный текстовый файл `.py`, который запускают командой `python имя_файла.py`.
- VS Code и PyCharm — два основных инструмента этой книги для написания и запуска кода.
- Интерактивная оболочка (Python Shell) выполняет код построчно и сразу показывает результат — удобно как калькулятор.
- `print()` умеет выводить несколько значений сразу и

принимает параметры `sep` и `end`.

- IDLE — простой редактор, который устанавливается вместе с Python; полезно знать, но для реальных проектов удобнее VS Code или PyCharm.

ГЛАВА 4 · СТР. 39

Python любит числа

Переменные, комментарии, три вида чисел и преобразование между ними.

Числа в Python

Числа — самый естественный вид данных для компьютера. Разберёмся, как их сохранять в переменные и как их правильно называть.

Числа в Python

Числа — один из самых часто используемых видов данных в программировании: возраст, цена, счёт в игре, координаты на экране. Python умеет работать с числами «из коробки», без каких-либо дополнительных приготовлений.

chisla.py

```
print(7)
print(3.5)
print(7 + 3.5)
```

Сохраняем числа

Чтобы использовать число не один раз, а много, его удобно сохранить в **переменную** — именованную «коробку» в памяти компьютера. Значение записывают знаком `=`:

peremennye.py

```
age = 10
print(age)

age = 11  # значение можно заменить в любой момент
print(age)
```

Правила именования переменных

- Имя может содержать буквы, цифры и знак подчёркивания `_`, но не может **начинаться** с цифры.
- Python различает регистр: `age` и `Age` — разные переменные.
- Нельзя использовать зарезервированные слова языка — например, `for`, `if`, `print` лучше тоже не использовать как имя переменной, хотя формально это не ключевое слово.
- По соглашению имена переменных в Python пишут в стиле `snake_case`: словами через нижнее

подчёркивание, например `user_age`, а не `userAge`.

Частая ошибка новичков

`1_place = "Cartesian"` — SyntaxError: имя переменной не может начинаться с цифры. Решение: переставьте слова местами — `place_1`.

Без подсказки типа → с подсказкой типа (Python 3.14)

Классический подход

```
age = 10
price = 19.99
name = "Cartesian"
# тип виден только по значению
```

Современный Python 3.14

```
age: int = 10
price: float = 19.99
name: str = "Cartesian"
# тип виден прямо в коде — подсказка (type hint)
```

Что использовать сегодня: и то, и другое работает одинаково — подсказки типов не влияют на выполнение программы. В маленьких учебных примерах классический вариант проще и короче, поэтому в первых главах книги мы обычно пишем без подсказок. Но как только код вырастает, подсказки типов делают его понятнее — редакторы вроде VS Code и PyCharm используют их, чтобы подсвечивать ошибки ещё до запуска программы. Мы вернёмся к ним подробнее в главе 13.

Комментарии

Текст в коде, который читает только человек — компьютер его полностью пропускает.

Комментарий — это текст в коде, который Python полностью игнорирует при выполнении. Он нужен не компьютеру, а человеку: объяснить, зачем нужна эта строка, оставить напоминание себе или коллеге.

kommentarii.py

```
# Это комментарий — Python его не выполняет
age = 10  # а это комментарий в конце строки
print(age)
```

Комментарий начинается со знака `#` и продолжается до конца строки.

> **Хороший комментарий объясняет «почему», а не «что»**
>
> Строка `age = 10 # присваиваем age значение 10` — бесполезный комментарий: и так понятно из кода. А `age = 10 # минимальный возраст по умолчанию` объясняет то, чего в самом коде не видно.

В этой книге мы придерживаемся именно такого стиля: комментарии на русском языке объясняют идею, а не пересказывают код построчно.

Числа бывают разных видов

Python различает целые числа, числа с плавающей точкой и комплексные числа — и у каждого вида свои особенности.

В Python есть несколько встроенных видов («типов») чисел. Три основных вы встретите чаще всего:

Целые числа

Тип `int` (от *integer*) — числа без дробной части: `-3, 0, 42, 1000000`. В Python размер целого числа практически не ограничен — можно работать даже с числами из сотен цифр.

celye.py

```
big = 10 ** 20
print(big)
print(type(big))
```

Числа с плавающей точкой

Тип `float` — числа с дробной частью: `3.14, -0.5, 2.0`. Даже если дробная часть равна нулю, точка делает число `float`, а не `int`.

plavayuschie.py

```
pi = 3.14
print(type(pi))

whole = 2.0
print(type(whole))  # float, несмотря на нулевую дробную часть
```

Деление всегда даёт float

Оператор `/` в Python всегда возвращает `float`, даже если числа делятся нацело: `10 / 2` равно `5.0`, а не `5`. Мы уже видели это в главе 3.

Комплексные числа

Тип `complex` — числа вида «действительная часть + мнимая часть», где мнимая часть отмечена буквой `j`. Такие числа нужны в основном в инженерных и научных расчётах (например, при обработке сигналов) — в большинстве обычных программ вы их не встретите, но полезно знать, что Python поддерживает их «из коробки».

kompleksnye.py

```
z = 3 + 4j
print(z)
print(type(z))
print(z.real, z.imag)
```

Преобразование типов чисел

Числа можно превращать друг в друга и в текст — но не всегда наоборот, и не всегда так, как кажется на первый взгляд.

У каждого числового типа есть своя функция-преобразователь с тем же именем: `int()`, `float()`, `complex()`. Кроме того, любое число можно

превратить в текст функцией `str()` — и наоборот, текст, похожий на число, можно превратить обратно.

preobrazovanie.py

```
whole = int(3.99)
print(whole)      # 3 — дробная часть
отбрасывается, а не округляется

decimal = float(7)
print(decimal)    # 7.0

text = str(42)
print(text, type(text))   # "42" <class
'str'>
```

int() не округляет, а отбрасывает дробную часть

`int(3.99)` равно `3`, а не `4`. Для настоящего округления нужна отдельная функция `round()` — о ней в главе 5.

Преобразование текста в число

Если строка действительно выглядит как число, её можно превратить обратно:

tekst_v_chislo.py

```
age_text = "10"
age = int(age_text)
print(age + 5)   # 15 — теперь это настоящее число
```

Не любой текст получится преобразовать

`int("десять")` вызовет `ValueError` — Python не умеет читать числа словами. Преобразовать можно только текст, который выглядит как число: цифры, возможно со знаком и точкой.

Сборка строки из чисел: конкатенация → f-строка

Классический подход

```
age = 10
message = "Возраст: " + str(age) + " лет"
print(message)
# обязательно вручную оборачивать число в str()
```

Современный Python 3.14

```
age = 10
message = f"Возраст: {age} лет"
print(message)
# f-строка сама выполняет преобразование внутри {}
```

Что использовать сегодня: f-строки (f-strings), появившиеся в Python 3.6 и с тех пор ставшие стандартом. Они короче, меньше подвержены ошибкам (не нужно помнить о str()) и позволяют сразу писать выражения внутри фигурных скобок. Классический способ через + всё ещё встречается в старом коде и стоит уметь его читать, но для нового кода используйте f-строки.

Мини-проект — Понимаете ли вы числа?

Небольшая проверка понимания — без единого запуска кода, только рассуждением — и краткие итоги главы.

Проверим, насколько хорошо вы понимаете числа в Python — без запуска кода, только рассуждением. Ответы — в ноутбуке практики.

★ Базовая практика

Угадайте тип

Определите тип каждого значения, не запуская код: `10`, `10.0`, `10 / 2`, `10 // 2`, `"10"`.

★★ Самостоятельная задача

Угадайте результат

Что выведет `print(int(7.9))`? А `print(str(7) + str(9))` — одно число 16 или что-то другое?

★★★ Задача повышенной сложности

Найдите ошибку

Строка `total = "Итого: " + 100` вызывает `TypeError`. Почему, и как её исправить двумя разными способами (через `str()` и через f-строку)?

Итоги

Что мы узнали в этой главе

- Числа сохраняют в **переменных** знаком `=`; имена переменных пишут в стиле `snake_case` и не начинают с цифры.
- `# комментарий` — текст, который Python игнорирует; он объясняет код человеку.
- Три числовых типа: `int` (целые), `float` (дробные), `complex` (комплексные, для науки и инженерии).
- `int()`, `float()`, `str()` преобразуют значения между типами — но `int()`

отбрасывает дробную часть, а не округляет.

- f-строки (`f"...{значение}"`) — современный и самый удобный способ вставлять числа в текст.

ГЛАВА 5 · СТР. 59

Давайте поиграем с числами!

Математические операторы, порядок вычислений, модули math и random.

Основные математические операции

Четыре оператора, знакомые со школы — но у деления в Python есть особенность, которую мы уже видели в главе 3.

В главе 4 мы познакомились с числами. Теперь научим Python по-настоящему с ними работать — считать, сравнивать, комбинировать. Начнём с четырёх операций, знакомых из школьной математики.

osnovnye_operacii.py

```
print(5 + 3)   # сложение
print(5 - 3)   # вычитание
print(5 * 3)   # умножение
print(5 / 3)   # деление — всегда float
```

Числа и переменные — как в главе 4

Все эти операторы работают и с переменными, не только с числами напрямую: `a + b` работает точно так же, как `5 + 3`, если `a` и `b` — числа.

Специальные математические операции в Python

Три оператора, которых нет на обычном калькуляторе — но без них не обходится почти ни одна программа.

Кроме четырёх основных операций, у Python есть три специальных, которых нет в привычном калькуляторе, но которые невероятно полезны в программировании.

Целочисленное деление `//`

Возвращает только целую часть результата деления, отбрасывая остаток:

celochislennoe.py

```
print(17 // 5)   # 3 — сколько раз 5 помещается в 17 целиком
print(17 / 5)
# 3.4 — обычное деление, для сравнения
```

Остаток от деления `%`

Возвращает то, что *осталось* после целочисленного деления. Один из самых полезных операторов в программировании — например, для проверки чётности числа:

ostatok.py

```
print(17 % 5)    # 2 — именно столько осталось
print(10 % 2)    # 0 — если остаток 0, число чётное
print(11 % 2)    # 1 — а если 1 — нечётное
```

Проверка чётности — классический приём

`число % 2 == 0` — самый частый способ проверить, чётное ли число. Мы воспользуемся им в мини-проекте главы 9.

Возведение в степень `**`

stepen.py

```
print(2 ** 10)    # 1024
print(9 ** 0.5)
# 3.0 — дробная степень 0.5 = квадратный корень
```

Операции присваивания

Более короткая запись для изменения переменных — и правила о том, что выполняется раньше, а что позже.

Операции присваивания

Изменить переменную «на основе самой себя» — очень частое действие: например, увеличить счёт в игре. Писать `score = score + 10` можно, но Python предлагает более короткую запись:

prisvaivanie.py

```
score = 0
score += 10   # то же самое, что score = score + 10
print(score)  # 10

score -= 3    # score = score - 3
print(score)  # 7
```

Такие сокращения существуют для всех основных операторов: `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`.

Что выполняется первым?

Как и в математике, у операторов в Python есть порядок выполнения: сначала — степень, затем — умножение, деление, целочисленное деление и остаток (слева направо), и в конце — сложение и вычитание.

poryadok.py

```
print(2 + 3 * 4)
# 14 — сначала 3 * 4 = 12, потом + 2
print((2 + 3) * 4)
# 20 — скобки меняют порядок
print(2 ** 3 ** 2)
# 512 — возведение в степень выполняется
справа налево: 2 ** (3 ** 2)
```

Сомневаетесь — используйте скобки

Даже когда порядок формально понятен, скобки делают код честнее для человека, который будет его читать (в том числе для вас через полгода). `(a + b) * c` читается быстрее, чем приходится вспоминать правила приоритета.

Интересные возможности работы с числами

Модуль math открывает пол и потолок числа, корни, факториалы и тригонометрию — всё то, чего не хватает базовым операторам.

Для более сложной математики в Python есть встроенный модуль `math` — набор готовых функций, который нужно сначала подключить командой `import math`.

Пол и потолок числа

Пол (floor) — ближайшее целое число снизу, **потолок** (ceil) — ближайшее целое число сверху.

pol_potolok.py

```
import math

print(math.floor(4.3))   # 4
print(math.floor(4.9))   # 4 — всегда вниз, даже если дробь большая
print(math.ceil(4.1))    # 5 — всегда вверх, даже если дробь маленькая
```

А как же обычное округление?

Для округления «как в школе» (0.5 и выше — вверх) в Python есть встроенная функция `round()` — её не нужно импортировать из `math`: `round(4.5)` → `4` (осторожно: из-за особого банковского округления Python иногда округляет «.5» к ближайшему чётному числу).

Степень и квадратный корень

koren.py

```
import math

print(math.pow(2, 10))   # 1024.0 — всегда возвращает float, в отличие от **
print(math.sqrt(81))     # 9.0
```

Факториал числа

Факториал числа `n` (обозначается `n!`) — произведение всех целых чисел от 1 до `n`. Используется, например, для подсчёта числа возможных перестановок.

faktorial.py

```
import math

print(math.factorial(5))
# 5! = 1 * 2 * 3 * 4 * 5 = 120
```

Синус, косинус, тангенс и многое другое

Модуль `math` включает и тригонометрические функции — они понадобятся, например, при рисовании окружностей и дуг в главе 7.

trigonometriya.py

```
import math

print(math.sin(math.pi / 2))   # 1.0
print(math.cos(0))             # 1.0
print(math.tan(math.pi / 4))   #
приблизительно 1.0
```

Другие числовые операции

Ещё несколько полезных инструментов, которые не требуют импорта: встроенные функции `abs()` (модуль числа), `min()` и `max()` (наименьшее и наибольшее из нескольких значений):

drugie.py

```
print(abs(-7))          # 7
print(min(4, 9, 2))     # 2
print(max(4, 9, 2))     # 9
```

Работа со случайными числами

Без капли случайности сложно представить хоть одну игру — знакомимся с модулем random.

Модуль `random` добавляет в Python элемент случайности — без него не обходится почти ни одна игра:

случайный противник, случайная карта, случайное число для игры «Угадай число» в главе 9.

sluchaynye.py

```
import random

print(random.random())          # случайное дробное число от 0.0 до 1.0 (не включая 1.0)
print(random.randint(1, 6))     # случайное целое от 1 до 6 включительно — как кубик
print(random.uniform(1, 10))    # случайное дробное от 1 до 10
```

randint включает обе границы

`random.randint(1, 6)` может вернуть и `1`, и `6` — обе границы включены. Это частая причина ошибок «на единицу» (off-by-one), если забыть об этом.

Случайный выбор из списка

Забегая немного вперёд (списки подробно разберём в главе 11) — `random.choice()` умеет выбрать случайный элемент из готового набора значений:

sluchaynyj_vybor.py

```
import random

variants = ["камень", "ножницы",
"бумага"]
print(random.choice(variants))
```

Диапазон случайных чисел: вручную → готовая функция

Классический подход

```
import random

# случайное целое от 1 до 6 вручную:
roll = int(random.random() * 6) + 1
print(roll)
```

Современный Python 3.14

```
import random

# то же самое — готовой функцией
roll = random.randint(1, 6)
print(roll)
```

Что использовать сегодня: готовые функции вроде `randint()` и `uniform()` — они существовали в модуле random ещё в старых версиях Python, но начинающие часто сначала пытаются собрать диапазон вручную через `random.random()`, не зная о них. Готовые функции короче и меньше подвержены ошибкам — особенно на границах диапазона.

Мини-проект — кратные числа

Собираем операторы главы вместе в одной небольшой программе — и подводим итоги.

Соберём операторы этой главы в один мини-проект: программу, которая находит все числа, кратные заданному, в диапазоне.

kratnye_chisla.py

```
import random

# случайное число, кратности которого мы ищем
kratnoe_chemu = random.randint(2, 9)
print(f"Ищем числа, кратные {kratnoe_chemu}, от 1 до 50")

chislo = 1
while chislo <= 50:
    if chislo % kratnoe_chemu == 0:
        print(chislo)
    chislo += 1
```

Забегаем вперёд

В этом примере использован цикл `while` — мы разберём его подробно в главе 10. Здесь достаточно увидеть, как оператор `%` из этой главы находит все числа, кратные заданному: остаток от деления равен нулю именно тогда, когда число делится нацело.

★ Базовая практика

Кратные тройке

Измените программу так, чтобы она всегда искала числа, кратные 3, без `random`.

★★ Самостоятельная задача

Свой диапазон

Измените границы поиска на 1–100 вместо 1–50.

★★★ Задача повышенной сложности

Считаем количество

Добавьте переменную-счётчик, которая считает, сколько кратных чисел было найдено, и выведите её значение в конце.

Итоги

Что мы узнали в этой главе

- Основные операторы: `+ - * /`; специальные: `// % **`.
- `%` (остаток от деления) — классический способ проверить чётность и кратность числа.
- Сокращённые операторы присваивания (`+=` и другие) изменяют переменную короче, чем `x = x + ...`.
- Порядок вычислений совпадает со школьной математикой; скобки — самый надёжный способ сделать порядок явным.

- Модуль `math` добавляет floor, ceil, sqrt, factorial и тригонометрию.
- Модуль `random` добавляет случайность — основу для игр из следующих глав.

ГЛАВА 6 · СТР. 83

Рисуем классные вещи с помощью Turtle

Первая графика на Python: движение, повороты, готовые фигуры и мандалы из прямых линий.

Приступаем

Знакомимся с двумя главными объектами графики Turtle: экраном и самой черепашкой.

Модуль `turtle` входит в стандартную поставку Python — ничего дополнительно устанавливать не нужно. Название отсылает к «черепашьей графике» (turtle graphics), придуманной ещё в 1960-х годах для обучения детей программированию — идея настолько удачная, что дожила до наших дней практически без изменений.

Экран и черепашка — два разных объекта

Прежде чем рисовать, нужны два объекта: **экран** (`turtle.Screen()`) — окно, в котором происходит рисование, и **черепашка** (`turtle.Turtle()`) — то, что, собственно, рисует.

nastroyka_ekrana.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

screen.exitonclick()  # держим окно открытым до клика мышью
```

Запустив этот код, вы увидите пустое белое окно с маленьким чёрным треугольником посередине — это и есть черепашка в исходном положении. Курс (направление) черепашки по умолчанию направлен вправо — на восток.

Зачем два отдельных объекта?

Разделение экрана и черепашки позволяет иметь на одном экране сразу **несколько** черепашек — мы воспользуемся этим в мини-проекте «Гонка Turtle» в главе 12.

Движение вперёд и назад

Первое, что должна уметь черепашка — двигаться. В этом разделе вы напишете свою первую программу, которая по-настоящему рисует.

Зачем это нужно

Всё, что вы нарисуете дальше — квадраты, шестиугольники, мандалы, даже персонажей в играх — строится из одной и той же пары действий: **«проехать немного»** и **«повернуть на**

какой-то угол». Освоив эти четыре команды, вы получите строительные блоки для всей главы.

Простая модель

Представьте черепашку как перо на листе бумаги. У пера есть две вещи: **позиция** (где оно сейчас находится) и **курс** — направление, в которое оно «смотрит». Когда черепашка едет вперёд, она просто перемещается по прямой в сторону своего курса. Когда она поворачивает, меняется курс, а позиция остаётся прежней — как будто вы стоите на месте и просто поворачиваетесь.

Синтаксис

У объекта черепашки есть четыре базовые команды движения:

- `forward(расстояние)` — проехать вперёд на указанное число пикселей (короткая форма — `fd`)
- `backward(расстояние)` — проехать назад, не поворачиваясь (короткие формы — `bk`, `back`)
- `right(угол)` — повернуть по часовой стрелке на угол в градусах (короткая форма — `rt`)
- `left(угол)` — повернуть против часовой стрелки на угол в градусах (короткая форма — `lt`)

Первый рабочий пример

Нарисуем букву «Г»: проедем вперёд, повернём и проедем ещё раз.

turtle_dvizhenie.py

```
import turtle

# создаём окно и черепашку
screen = turtle.Screen()
artist = turtle.Turtle()

# едем вперёд на 120 пикселей
artist.forward(120)

# поворачиваем на 90 градусов против
часовой стрелки
artist.left(90)

# едем вперёд ещё раз — получаем букву
«Г»
artist.forward(80)

screen.exitonclick()
# окно закроется по клику мыши
```

Разбор по шагам

- `turtle.Screen()` открывает окно для рисования.
- `turtle.Turtle()` создаёт саму черепашку — в начале координат, курс направлен вправо (на восток).
- `artist.forward(120)` — черепашка проезжает 120 пикселей в направлении курса, оставляя линию.
- `artist.left(90)` — курс поворачивается на 90° против часовой стрелки (теперь черепашка «смотрит» вверх).
- Второй вызов `forward()` едет уже в новом направлении — отсюда и получается уголок буквы «Г».
- `screen.exitonclick()` держит окно открытым, пока вы не кликнете по

нему — без этой строки окно может закрыться мгновенно.

> **Совет**
>
> Числа в `forward()` и `right()` / `left()` могут быть отрицательными. `forward(-50)` и `backward(50)` делают одно и то же.

Эксперимент 1

Измените расстояние 120 на 250 и угол 90 на 45. Перед запуском попробуйте предсказать: в какую сторону теперь «смотрит» черепашка после поворота?

Эксперимент 2

Добавьте третью пару команд `left(90)` и `forward(120)` в конец программы. Какая фигура получится?

Типичная ошибка

> **Внимание**
>
> Начинающие часто путают `right()` / `left()` с движением — кажется, что черепашка должна сместиться в сторону. На самом деле эти команды **только поворачивают** курс и не двигают черепашку ни на пиксель.

Диагностика: если после поворота фигура выглядит «сплющенной» или линии накладываются друг на друга —

скорее всего, вы забыли вызвать `forward()` после поворота, либо перепутали местами поворот и движение.

Исправление: перечитайте программу построчно и для каждой команды спросите себя: «это движение или поворот?» — держите эти два действия отдельно друг от друга.

Современный вариант

В старом коде черепашку часто рисуют без явного объекта — модуль `turtle` автоматически создаёт «черепашку по умолчанию», и её можно двигать напрямую через функции модуля.

Классический подход → современный Python 3.14

Классический подход

```
import turtle

# без явного объекта —
# используется скрытая
# черепашка по умолчанию
turtle.forward(120)
turtle.left(90)
turtle.forward(80)
turtle.done()
```

Современный Python 3.14

```
import turtle

# явный объект — понятно,
# какая черепашка рисует,
# легко добавить вторую
artist = turtle.Turtle()
artist.forward(120)
artist.left(90)
artist.forward(80)
turtle.done()
```

Что использовать сегодня: явный объект `turtle.Turtle()`. Он ничего не усложняет, зато сразу готовит вас к главе 14 (объекты) и легко масштабируется — если позже понадобится нарисовать гонку из нескольких черепашек (см. мини-проект в главе 12), у каждой будет своё имя и своё состояние.

Модульные функции (`turtle.forward()` без объекта) всё ещё встречаются в старом коде и учебниках — знать их полезно, но начинайте привычку сразу с объекта.

Задание

★ Базовая практика

Буква «Т»

Используя только `forward`, `backward`, `left` и `right`, нарисуйте букву «Т» одной непрерывной линией пера.

★★ Самостоятельная задача

Лестница из трёх ступенек

Нарисуйте лестницу, состоящую из трёх одинаковых «ступенек» (вперёд-поворот-вперёд-поворот). Попробуйте обойтись без повторения одинакового блока кода вручную — подсказка: этому вы научитесь в главе 10, а пока просто повторите блок трижды подряд.

Что запомнить

- `forward()` / `backward()` двигают черепашку по прямой; `left()` / `right()` только поворачивают курс.
- Позиция и курс — это два независимых свойства черепашки.
- Современный стиль — создавать явный объект `turtle.Turtle()`, а не полагаться на скрытую черепашку по умолчанию.
- `screen.exitonclick()` или `turtle.done()` нужны, чтобы окно не закрылось мгновенно.

Заставляем черепашку менять направление

Кроме относительных поворотов left()/right(), у черепашки есть способ задать направление абсолютно — вне зависимости от того, куда она смотрела раньше.

Команды `left()` / `right()` из предыдущего раздела поворачивают черепашку **относительно** её текущего курса. Иногда удобнее задать направление **абсолютно** — «смотри строго на север», независимо от того, куда черепашка смотрела раньше.

Абсолютный курс: `setheading()`

В Turtle направления измеряются в градусах против часовой стрелки, начиная с востока (0°): восток — 0°, север — 90°, запад — 180°, юг — 270°.

setheading.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

artist.setheading(90)   # смотрим строго вверх, неважно, куда смотрели раньше
artist.forward(100)

artist.setheading(180)  # смотрим строго влево
artist.forward(100)

screen.exitonclick()
```

Возврат домой: `home()`

Возвращает черепашку в исходную точку (0, 0) с исходным курсом (0°) — удобно, чтобы начать рисунок заново, не создавая нового объекта:

home.py

```
artist.home()
```

heading() — узнать текущий курс

Если нужно не задать курс, а *узнать* его — используйте `artist.heading()`: она возвращает текущее направление в градусах, не меняя его.

Мини-проект — рисуем квадрат

Первые настоящие фигуры — применяем движение и повороты из предыдущих разделов.

Применим движение и повороты, чтобы нарисовать первые настоящие фигуры.

Мини-проект — рисуем квадрат

У квадрата четыре одинаковые стороны и четыре угла по 90°. Значит, нужно четыре раза повторить одну и ту же пару команд: проехать вперёд, повернуть на 90°.

kvadrat.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)

screen.exitonclick()
```

Заметили повторение?

Четыре одинаковых блока подряд — явный признак того, что напрашивается цикл `for`. Мы вернёмся к этому же квадрату в главе 10 и перепишем его в 2 строки вместо 8.

Мини-проект — рисуем шестиугольник

У шестиугольника шесть равных сторон, и сумма всех внешних углов многоугольника всегда равна 360° — значит, угол поворота на каждом шаге равен `360 / 6 = 60` градусов.

shestiugolnik.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)

screen.exitonclick()
```

Формула для любого правильного многоугольника

Угол поворота = `360 / количество_сторон`. Треугольник — 120°, квадрат — 90°, пятиугольник — 72°, шестиугольник — 60°. Попробуйте эту формулу в ноутбуке практики на фигуре по вашему выбору.

Сокращённые приёмы

Каждая команда движения имеет более короткий псевдоним — полезно уметь читать оба варианта.

У каждой команды движения, которую мы изучили, есть более короткий псевдоним — они делают абсолютно то же самое, только короче печатать:

- `forward()` → `fd()`
- `backward()` → `bk()` (или `back()`)
- `left()` → `lt()`
- `right()` → `rt()`

Полные имена команд → короткие псевдонимы

Полные имена	Короткие псевдонимы
`artist.forward(100)`	`artist.fd(100)`
`artist.backward(50)`	`artist.bk(50)`
`artist.left(90)`	`artist.lt(90)`
`artist.right(90)`	`artist.rt(90)`

Что использовать сегодня: оба варианта официальны и делают ровно одно и то же — псевдонимы существуют в модуле turtle специально для более быстрого написания. В этой книге мы обычно используем полные имена, потому что они понятнее при первом чтении, но короткие формы стоит узнавать: в чужом коде и примерах в интернете вы будете встречать оба варианта.

Другие полезные сокращения

Помимо движения, часто используют ещё две короткие команды для оформления:

oformlenie.py

```
artist.pencolor("purple")   # цвет линии
artist.pensize(3)            # толщина линии
```

Переходим к случайным точкам на экране

goto() перемещает черепашку в любую точку экрана напрямую — вместе со случайными числами получается интересный эффект.

До сих пор черепашка двигалась только относительно своей текущей позиции. Но можно сразу «телепортировать» её в конкретную точку экрана командой `goto(x, y)` — а вместе со случайными числами из главы 5 это открывает интересные возможности.

Система координат экрана

Центр экрана Turtle — точка `(0, 0)`. Ось X растёт вправо, ось Y растёт вверх (в отличие от многих других графических систем, где Y растёт вниз — ещё один повод свериться с документацией, если работаете с другой библиотекой). По умолчанию окно имеет размер 400×300 точек в каждую сторону от центра.

sluchaynye_tochki.py

```
import turtle
import random

screen = turtle.Screen()
artist = turtle.Turtle()
artist.penup()  # поднимаем перо — при
движении линия не рисуется

for _ in range(10):
    x = random.randint(-200, 200)
    y = random.randint(-150, 150)
    artist.goto(x, y)
    artist.pendown()
    artist.dot(10)  # рисуем точку
диаметром 10
    artist.penup()

screen.exitonclick()
```

penup() / pendown()

Без `penup()` черепашка будет рисовать линию при каждом перемещении `goto()` — что обычно не то, что нужно для «прыжка» в новую точку. Не забывайте включить перо обратно `pendown()`, когда снова понадобится рисовать.

Рисуем квадрат с помощью goto

Тот же квадрат, что и раньше, — но на этот раз через прямое указание координат каждого угла.

Вернёмся к квадрату из раздела 6.4 — на этот раз нарисуем его без единого поворота, задавая напрямую четыре угловые точки командой `goto()`.

kvadrat_goto.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

artist.goto(100, 0)
artist.goto(100, 100)
artist.goto(0, 100)
artist.goto(0, 0)

screen.exitonclick()
```

Два способа — один результат

`forward()` + `right()` описывают движение *относительно черепашки* («проехать, повернуть»); `goto()` описывает движение *относительно экрана* («окажись в этой точке»). Иногда одно удобнее другого — например, goto() удобен, когда координаты фигуры уже известны заранее.

Мини-проект — рисуем мандалу только из прямых линий

Собираем все приёмы главы в одном узоре — и подводим итоги.

Соберём все приёмы главы в одном мини-проекте: мандалу, составленную только из прямых линий, — множество отрезков одинаковой длины, каждый раз повёрнутых на небольшой угол.

mandala.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()
artist.speed(0)  # максимальная скорость
рисования

shag_ugla = 10
ugol = 0
while ugol < 360:
    artist.setheading(ugol)
    artist.forward(150)
    artist.backward(150)
    ugol += shag_ugla

screen.exitonclick()
```

Забегаем вперёд

Цикл `while` подробно разберём в главе 10 — здесь достаточно увидеть, что `setheading()` из раздела 6.3 позволяет нарисовать сложный узор всего в нескольких строках, перебирая углы от 0 до 360.

★ Базовая практика

Другой шаг угла

Измените `shag_ugla` на 5 или на 30 — как меняется узор?

★★ Самостоятельная задача

Другая длина луча

Измените длину `forward(150)` — попробуйте 80 и 250.

★★★ Задача повышенной сложности

Цветная мандала

Добавьте `artist.pencolor("violet")` перед циклом и поэкспериментируйте с другими цветами.

Итоги

Что мы узнали в этой главе

- Экран (`turtle.Screen()`) и черепашка (`turtle.Turtle()`) — два разных объекта.
- `forward()` / `backward()` двигают черепашку; `left()` / `right()` поворачивают её курс относительно текущего направления.
- `setheading()` задаёт направление абсолютно, `home()` возвращает черепашку в начало координат.
- Угол поворота для правильного многоугольника — `360 / количество_сторон` .

- `goto(x, y)` перемещает черепашку в конкретную точку экрана; `penup()` / `pendown()` управляют тем, рисуется ли линия при движении.

ГЛАВА 7 · СТР. 107

Глубокое погружение в Turtle

Настройка экрана и графики, окружности, дуги, текст на экране и два новых мини-проекта.

Настраиваем экран

Прежде чем рисовать, настроим сам холст: заголовок окна, цвет фона и размер.

Прежде чем рисовать что-то сложнее квадратов, полезно настроить сам холст — размер окна, заголовок, цвет фона.

nastrojka_ekrana.py

```
import turtle

screen = turtle.Screen()
screen.title("Моя первая картина")     # заголовок окна
screen.bgcolor("lightblue")
# цвет фона
screen.setup(width=600, height=500)
# размер окна в пикселях

artist = turtle.Turtle()
screen.exitonclick()
```

Цвет можно задать и по-другому

Кроме названий вроде `"lightblue"`, Turtle понимает шестнадцатеричные коды: `screen.bgcolor("#0D0230")` — тот же формат, что мы используем для фирменных цветов Cartesian School на сайте книги.

Настраиваем графику

Скорость, толщина линии, цвет и форма черепашки — тонкая настройка перед рисованием.

Теперь настроим саму черепашку: скорость рисования, толщину и цвет линии, форму указателя.

nastrojka_grafiki.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

artist.speed(3)          # скорость от 1 (медленно) до 10 (быстро), 0 — мгновенно
artist.pensize(3)        # толщина линии в пикселях
artist.pencolor("purple") # цвет линии
artist.shape("turtle")    # форма указателя — черепашка вместо стрелки

artist.forward(100)
screen.exitonclick()
```

speed(0) — режим художника

Когда важен результат, а не сам процесс рисования (например, в сложных узорах вроде мандалы из главы 6), удобно поставить `artist.speed(0)` — черепашка рисует мгновенно, без анимации движения.

Фигуры без линий

Закрашенные фигуры, готовая команда для окружностей и точки без движения черепашки.

Фигуры без линий: заливка цветом

Чтобы нарисовать не просто контур, а закрашенную фигуру, оберните рисование между `begin_fill()` и `end_fill()` — цвет заливки задаётся заранее командой `fillcolor()`.

zalivka.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

artist.fillcolor("gold")
artist.begin_fill()
for _ in range(4):
    artist.forward(100)
    artist.right(90)
artist.end_fill()

screen.exitonclick()
```

Окружности

Рисовать окружность вручную через множество коротких отрезков не нужно — у Turtle есть готовая команда `circle(радиус)`:

okruzhnost.py

```
artist.circle(60)               # окружность радиусом 60
artist.circle(-60)              # отрицательный радиус — по часовой стрелке
```

Точки

Мы уже видели `dot()` в главе 6 — маленький закрашенный кружок в текущей позиции, без движения черепашки:

tochki.py

```
artist.dot(20, "red")
# точка диаметром 20, красная
```

Дуги

circle() умеет рисовать не только полные окружности, но и их части — дуги заданного размера.

Дуга — это часть окружности. У `circle()` есть второй, необязательный аргумент `extent` — сколько градусов окружности нужно нарисовать:

dugi.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

artist.circle(80, 180)   # половина
окружности (180°)
artist.circle(80, 90)    # четверть
окружности (90°)

screen.exitonclick()
```

Полукруг + прямая = купол

Комбинируя дугу с обычной линией, легко получить составные фигуры — например, полукруглый купол домика или арку. Попробуйте в ноутбуке практики.

Ещё больше возможностей!

Несколько приёмов на будущее: штампы, скрытие черепашки и отмена последнего действия.

Ещё несколько приёмов, которые пригодятся в будущих главах — особенно в играх.

Штамп: `stamp()`

Оставляет отпечаток текущей формы черепашки на экране, не рисуя линию — полезно для множества одинаковых объектов на экране (например, звёзд неба или врагов в игре):

stamp.py

```
artist.shape("circle")
artist.stamp()
artist.forward(50)
artist.stamp()
```

Скрыть и показать черепашку

skryt.py

```
artist.hideturtle()   # спрятать указатель черепашки (сама линия останется)
artist.showturtle()   # показать обратно
```

Зачем прятать черепашку?

В готовых рисунках указатель черепашки (маленький треугольник или другая форма) может визуально мешать. `hideturtle()` в конце программы делает финальный результат чище.

Отмена последнего действия: `undo()`

undo.py

```
artist.forward(100)
artist.undo()   # отменяет последнее движение, как Ctrl+Z
```

Рисуем текст на экране

Turtle умеет не только линии — научим её подписывать собственные рисунки.

Turtle умеет не только рисовать линии, но и выводить текст прямо на холсте — командой `write()`:

tekst.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()

artist.write("Привет, Turtle!",
font=("Arial", 20, "normal"))

screen.exitonclick()
```

Параметр `font` — кортеж из трёх значений: название шрифта, размер и начертание (`"normal"`, `"bold"` или `"italic"`).

Текст можно выровнять

Необязательный параметр `align` (`"left"`, `"center"` или `"right"`) управляет тем, как текст выравнивается относительно текущей позиции черепашки.

Мини-проект — окружность внутри квадрата

Первая композиция из нескольких фигур — квадрат и точно вписанная в него окружность.

Соберём фигуры этой главы в одном рисунке: квадрат с вписанной в него окружностью.

okruzhnost_v_kvadrate.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()
artist.speed(0)

razmer = 150

# квадрат
for _ in range(4):
    artist.forward(razmer)
    artist.right(90)

# перемещаемся к точке, откуда начнётся
вписанная окружность
artist.penup()
artist.goto(razmer / 2, -razmer / 2)
artist.setheading(0)
artist.pendown()
artist.circle(razmer / 2)

screen.exitonclick()
```

Откуда взялась эта точка?

Чтобы окружность радиусом `razmer / 2` идеально вписалась в квадрат, черепашка должна начать рисовать её из точки на полпути вдоль нижней стороны — отсюда и вычисление координат через `penup()` / `goto()` из главы 6.

★★ Самостоятельная задача

Окружность в шестиугольнике

Замените квадрат на шестиугольник из главы 6 и впишите в него окружность подходящего радиуса.

Меняем направление рисования

Отрицательный радиус и режимы измерения углов — ещё немного тонкой настройки перед финальным мини-проектом главы.

По умолчанию `circle()` рисует окружность против часовой стрелки. Отрицательный радиус меняет направление на противоположное — по часовой стрелке:

napravlenie.py

```
artist.circle(60)    # против часовой
стрелки
artist.circle(-60)   # по часовой стрелке
```

Режимы измерения углов

По умолчанию Turtle измеряет углы в градусах. Если по какой-то причине удобнее работать в радианах (например, для совместимости с формулами из модуля `math` из главы 5) — есть команды `degrees()` и `radians()`:

radiany.py

```
import math

artist.radians()
artist.left(math.pi / 2)   # поворот на
90°, выраженный в радианах

artist.degrees()            #
возвращаемся к привычным градусам
```

Мини-проект — смайлик

Собираем все приёмы главы в одном дружелюбном рисунке — и подводим итоги.

Финальный мини-проект главы: нарисуем простой смайлик, используя всё, что узнали — окружности, заливку, дуги и точки.

smajlik.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()
artist.speed(0)

# лицо
artist.fillcolor("yellow")
artist.begin_fill()
artist.circle(100)
artist.end_fill()

# глаза
artist.penup()
artist.goto(-35, 120)
artist.pendown()
artist.dot(20, "black")
artist.penup()
artist.goto(35, 120)
artist.pendown()
artist.dot(20, "black")

# улыбка — нижняя дуга окружности
artist.penup()
artist.goto(-50, 60)
artist.setheading(-60)
```

```
artist.pendown()
artist.pensize(4)
artist.circle(60, 120)

artist.hideturtle()
screen.exitonclick()
```

★ Базовая практика

Другое настроение

Измените дугу улыбки на дугу нахмуренных бровей — переверните направление extent.

★★ Самостоятельная задача

Цветной смайлик

Смените fillcolor лица на любой другой цвет.

★★★ Задача повышенной сложности

Щёчки

Добавьте два маленьких розовых dot() под глазами — получатся румяные щёчки.

Итоги

Что мы узнали в этой главе

- `begin_fill()` / `end_fill()` закрашивают фигуру выбранным цветом.
- `circle(радиус, extent)` рисует окружности и дуги; отрицательный радиус меняет направление на противоположное.
- `write()` выводит текст прямо на холсте.
- `stamp()`, `hideturtle()` и `undo()` — полезные инструменты для более сложных рисунков.
- Композиция из нескольких фигур (квадрат + окружность, смайлик) строится

последовательным применением уже знакомых команд.

ГЛАВА 8 · СТР. 137

Играем с буквами и словами

Строки в Python: создание, индексы и срезы, методы, форматирование, ввод от пользователя.

Что такое строки?

Текст в Python — это строки. Разберёмся, как их создавать: в одну строку и в несколько сразу.

Что такое строки?

Строка (`str`) — это текст: последовательность символов — букв, цифр, знаков препинания, пробелов — заключённая в кавычки. Мы уже пользовались строками с самой первой программы в главе 1, теперь разберём их подробно.

Создаём строки

sozdaem_stroki.py

```
greeting = "Привет"
name = 'Cartesian'
print(greeting, name)
```

Одинарные и двойные кавычки в Python равнозначны — выбирайте любые, главное быть последовательным в рамках одной строки.

Хочу много-много строк!

Если текст должен занимать несколько строк, используют **тройные кавычки** — тогда переносы строк внутри текста сохраняются как есть:

mnogo_strok.py

```
poem = """Код за кодом,
шаг за шагом —
так рождается
программа."""
print(poem)
```

В моей строке есть кавычки! :O

Что делать с кавычками внутри строки — и как объединять несколько строк в одну.

В моей строке есть кавычки! :O

Что делать, если внутри текста нужна кавычка того же типа, что обрамляет строку? Самый простой способ — обрамить строку кавычками **другого** типа:

kavychki.py

```
quote = "Она сказала: 'Привет!'"
quote2 = 'Она сказала: "Привет!"'
print(quote)
print(quote2)
```

Если нужны именно такие же кавычки, как обрамляющие, — их экранируют обратной косой чертой `\`:

ekranirovanie.py

```
quote = "Она сказала: \"Привет!\""
print(quote)
```

Объединяем две или несколько строк

Строки, как и числа, можно складывать оператором `+` — это называется **конкатенацией**:

konkatenaciya.py

```
first_name = "Ада"
last_name = "Лавлейс"
full_name = first_name + " " + last_name
print(full_name)
```

Строку с числом напрямую не сложить

`"Возраст: " + 10` вызовет `TypeError` — мы уже видели это в главе 4. Нужно либо `str(10)`, либо f-строка (раздел 8.6).

Конкатенация в print()

Напомним: у `print()` есть более простой способ вывести несколько значений — через запятую, без явного `+` (мы делали это в главе 3):

print_konkatenaciya.py

```
print(first_name, last_name)          # через запятую — сам добавит пробел
print(first_name + " " + last_name)   # через + — нужно добавлять пробел самому
```

Пустая строка

Строка может не содержать ни одного символа — это тоже допустимая, «пустая» строка `""`. Она отличается от отсутствия значения и часто используется как стартовое значение перед накоплением текста.

Доступ к символам строки

Каждый символ строки можно достать по номеру — а срезы позволяют достать сразу целый кусок.

Доступ к символам строки

Строка — это последовательность символов, и к каждому символу можно обратиться по его **индексу** (порядковому номеру) в квадратных скобках. Индексация в Python начинается с **нуля**:

indeksy.py

```
word = "Python"
print(word[0])   # P — первый символ
print(word[1])   # y — второй символ
print(word[5])   # n — шестой (и последний) символ
```

Отрицательные индексы

Отрицательные индексы отсчитываются с конца строки — `-1` означает «последний символ»:

otricatelnye_indeksy.py

```
word = "Python"
print(word[-1])   # n — последний символ
print(word[-2])   # o — предпоследний
```

IndexError — индекс за пределами строки

`word[10]` для шестибуквенного слова вызовет `IndexError: string index out of range`. Индексы существуют только от `0` до `len(word) - 1` (и от -1 до -len(word) с конца).

Получение среза строки

Срез (slice) возвращает часть строки — от одного индекса до другого (не включая последний):

srezy.py

```
word = "Python"
print(word[0:3])   # Pyt — символы с индексами 0, 1, 2
print(word[2:])    # thon — от индекса 2 и до конца
print(word[:3])    # Pyt — от начала до индекса 3 (не включая)
print(word[::-1])  # nohtyP — весь текст задом наперёд!
```

Разворот строки одним срезом

`строка[::-1]` — самый короткий способ развернуть строку задом наперёд. Третий параметр среза — «шаг»; шаг -1 проходит строку в обратном порядке. Мы воспользуемся этим в мини-проекте 8.9.

Методы строк — магия работы со строками!

Готовые действия над строками, вызываемые через точку — от смены регистра до разбиения на слова.

У каждой строки в Python есть встроенные **методы** — готовые действия, которые можно выполнить над ней через точку: `строка.метод()`.

Верхний и нижний регистр

registr.py

```
text = "Python с нуля"
print(text.upper())   # ПРОГРАММИРОВАНИЕ КАПСОМ
print(text.lower())   # питон с нуля
print(text.title())   # Python С Нуля — Первая Буква Каждого Слова
```

Разные методы

raznye_metody.py

```
text = "  Python с нуля  "
print(text.strip())              # убирает пробелы по краям
print(text.replace("нуля", "начала"))  # замена подстроки
print(text.count("н"))           # сколько раз встречается символ
print("Python".startswith("Py"))  # True
print("Python".endswith("on"))    # True
print(text.split())              # разбивает строку на список слов
```

Методы строк не меняют исходную строку

Строки в Python **неизменяемы** — `text.upper()` возвращает *новую* строку, а исходная переменная `text` остаётся прежней, если не переприсвоить: `text = text.upper()`.

Истина? Ложь?

Сравниваем строки друг с другом и проверяем, входит ли одна строка в другую.

Строки можно сравнивать друг с другом и проверять, содержится ли одна строка внутри другой — результатом всегда будет `True` или `False`.

istina_lozh.py

```
print("Python" == "Python")     # True — строки совпадают полностью
print("Python" == "python")
# False — регистр важен
print("thon" in "Python")       # True — "thon" встречается внутри "Python"
print("java" in "Python")       # False
```

== сравнивает значение, is сравнивает объект

Для строк почти всегда нужен именно `==`. Оператор `is` проверяет, один ли это объект в памяти, — гораздо более редкий и специфичный случай, к которому мы вернёмся в главе 14.

Пустая строка — это «ложь»

В логических проверках (например, `if` — подробно в главе 9) пустая строка ведёт себя как `False`, а любая непустая строка — как `True`:

pustaya_stroka_logika.py

```
print(bool(""))          # False
print(bool("Python"))    # True
print(bool(" "))
# True — пробел — это тоже символ!
```

Форматирование строк

Три поколения одного и того же инструмента — вставить значение внутрь текста.

Мы уже пользовались f-строками начиная с главы 4 — теперь разберём форматирование строк полностью,

включая исторический контекст: в старом коде вы неизбежно встретите и другие способы.

Форматирование строк: % → .format() → f-строки

Классический подход (% и .format())

```
name = "Cartesian"
age = 5

# оператор % — самый старый способ
print("Привет, %s! Тебе %d лет." % (name, age))

# .format() — более новый, но всё ещё многословный
print("Привет, {}! Тебе {} лет.".format(name, age))
```

Современный Python 3.14 (f-строки)

```
name = "Cartesian"
age = 5

# f-строка — значения прямо внутри {}
print(f"Привет, {name}! Тебе {age} лет.")

# работают и выражения, не только переменные:
print(f"Через год будет {age + 1}.")
```

Что использовать сегодня: f-строки — они появились в Python 3.6 и с тех пор являются стандартом. Оператор `%`

— самый старый способ, доставшийся от языка Си, всё ещё встречается в старом коде. `.format()` — промежуточный шаг между ними. f-строки читаются лучше всего: значение видно прямо там, где оно будет подставлено, и можно писать любые выражения, а не только имена переменных.

Форматирование чисел внутри f-строк

Внутри фигурных скобок f-строки можно не только подставить значение, но и указать, как именно его отформатировать — например, сколько знаков после запятой:

format_specifikatory.py

```
pi = 3.14159265
print(f"Пи округлённо: {pi:.2f}")   # Пи
округлённо: 3.14
print(f"{1234567:,}")               #
1,234,567 — разделитель тысяч
```

Получение ввода от пользователей

Начало автоматизации: программа, которая спрашивает — и реагирует на ответ.

До сих пор все данные в программах были «зашиты» прямо в код. Пора научиться получать данные от самого пользователя во время выполнения

программы — с этого момента ваши программы становятся по-настоящему интерактивными.

vvod.py

```
name = input("Как вас зовут? ")
print(f"Привет, {name}!")
```

`input()` останавливает программу и ждёт, пока пользователь наберёт текст и нажмёт Enter — то, что он ввёл, возвращается как обычная строка.

В ноутбуке практики input() выглядит немного иначе

Jupyter Notebook выполняется автоматически, без живого человека за клавиатурой, поэтому в ноутбуке практики `input()` подставляет заранее заготовленные ответы — сам код при этом выглядит и работает точно так же, как в обычном `.py`-файле, который вы запускаете сами.

Преобразование строки в int или float

`input()` **всегда** возвращает строку — даже если пользователь ввёл число. Чтобы посчитать что-то с этим значением, его нужно преобразовать (как мы делали в главе 4):

vvod_chisla.py

```
age_text = input("Сколько вам лет? ")
age = int(age_text)
print(f"Через 5 лет вам будет {age +
5}.")
```

Частая ошибка новичков

Если забыть про `int()` и написать `age + 5`, где `age` — строка из `input()`, Python выдаст `TypeError`: складывать строку и число напрямую нельзя.

Мини-проект — выводим текст Turtle на новый уровень!

Объединяем input() из предыдущего раздела с write() из главы 7 — персональное приветствие прямо на холсте.

Соединим строки, ввод от пользователя и Turtle из глав 6–7: спросим имя и выведем персональное приветствие прямо на холсте.

turtle_tekst_imya.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()
artist.hideturtle()

name = input("Как вас зовут? ")

artist.penup()
artist.goto(0, 0)
artist.write(f"Привет, {name}!",
align="center", font=("Arial", 24,
"bold"))

screen.exitonclick()
```

★★ Самостоятельная задача

Приветствие по времени суток

Спросите у пользователя число (час дня) через input() + int(), и выведите разный текст в зависимости от значения — используя пока лишь то, что уже знаете о строках (полноценный if будет в главе 9, здесь достаточно вывести один текст с подставленным часом).

Мини-проект — кричим на экран

Два коротких, но показательных мини-проекта на методы и срезы строк.

Мини-проект — кричим на экран

Простая, но забавная программа: превращает любую фразу в «крик» — капсом и с кучей восклицательных знаков.

krik.py

```
phrase =
input("Что вы хотите прокричать? ")
krik = phrase.upper() + "!!!"
print(krik)
```

Мини-проект — переворачиваем своё имя

Используем срез `[::-1]` из раздела 8.3, чтобы развернуть введённое имя задом наперёд:

perevorot_imeni.py

```
name = input("Как вас зовут? ")
perevernutoe = name[::-1]
print(f"Ваше имя задом наперёд:
{perevernutoe}")
```

★ Базовая практика

Крик с ограничением

Измените krik.py так, чтобы восклицательных знаков было столько же, сколько букв во фразе (подсказка: умножение строки на число — str * n — повторяет её n раз).

★★ Самостоятельная задача

Палиндром?

Допишите perevorot_imeni.py так, чтобы он сообщал, является ли введённое слово палиндромом — читается ли оно одинаково в обе стороны (сравните name с перевёрнутой версией).

Мини-проект — красочная и динамическая математика

Собираем главу воедино: ввод пользователя, числа, строки и Turtle — в одном интерактивном мини-калькуляторе.

Финальный мини-проект главы объединяет практически всё: ввод от пользователя, числа из глав 4–5, строки и Turtle из глав 6–7 — маленький «калькулятор с характером».

dinamicheskaya_matematika.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()
artist.hideturtle()

a = float(input("Первое число: "))
b = float(input("Второе число: "))

artist.penup()
artist.goto(0, 60)
artist.pencolor("purple")
artist.write(f"{a} + {b} = {a + b}",
align="center", font=("Arial", 18,
"bold"))

artist.goto(0, 0)
artist.pencolor("blue")
artist.write(f"{a} * {b} = {a * b}",
align="center", font=("Arial", 18,
"bold"))

screen.exitonclick()
```

★★★ Задача повышенной сложности

Ещё две операции

Добавьте на холст ещё две строки: результат вычитания и деления — своим цветом для каждой.

Итоги

Что мы узнали в этой главе

- Строки создают в одинарных, двойных или тройных кавычках; тройные сохраняют переносы строк.
- Оператор `+` объединяет строки (конкатенация); `print(a, b)` — более простая альтернатива для вывода.
- Символы строки доступны по индексу (с нуля) и отрицательному индексу (с конца); срез `[a:b]` достаёт часть строки, `[::-1]` — разворачивает её.
- У строк десятки готовых методов: `upper()`, `lower()`,

`strip()`, `replace()`, `split()` и другие — все они возвращают новую строку, не меняя исходную.

- f-строки — современный способ форматирования; `%` и `.format()` — более старые, но всё ещё встречаются.
- `input()` всегда возвращает строку — для чисел нужно явное преобразование `int()` / `float()`.

ГЛАВА 9 · СТР. 175

Выполняй мою команду!

Логические значения, сравнения, условный оператор if/elif/else и первая настоящая игра.

Истина или ложь

Прежде чем программа сможет принимать решения, ей нужен способ хранить сам результат решения — логический тип bool.

До сих пор все наши программы выполняли одни и те же команды при каждом запуске. Пора научить их

принимать решения — делать одно, если верно одно условие, и другое, если верно другое. Всё начинается с логического типа данных.

Логический тип: bool

Тип `bool` имеет всего два возможных значения: `True` (истина) и `False` (ложь) — с заглавной буквы, это ключевые слова Python, а не обычный текст.

bool.py

```
is_sunny = True
is_raining = False
print(is_sunny, type(is_sunny))
```

Как получить bool

Чаще всего `True` / `False` не пишут вручную, а получают в результате сравнения:

sravnenie_bool.py

```
print(5 > 3)     # True
print(5 == 3)    # False
```

«Истинность» других типов

Мы уже видели в главе 8, что пустая строка ведёт себя как `False`, а непустая — как `True`. То же правило работает для чисел и для других типов данных, которые мы изучим позже:

istinnost.py

```
print(bool(0))      # False — ноль считается «ложью»
print(bool(42))     # True — любое ненулевое число — «истина»
print(bool(""))     # False — пустая строка
print(bool("нет"))  # True — любая непустая строка, даже такая!
```

bool("False") — это True!

Строка `"False"` — непустая, значит, `bool("False")` равно `True`. Единственная строка, которая ведёт себя как ложь — пустая строка `""`.

Сравниваем и принимаем решение

Шесть операторов, которые превращают два значения в один bool.

Полный набор операторов сравнения в Python:

operatory_sravneniya.py

```
print(5 == 5)   # равно
print(5 != 3)   # не равно
print(5 > 3)    # больше
print(5 < 3)    # меньше
print(5 >= 5)   # больше или равно
print(5 <= 3)   # меньше или равно
```

= против ==

Одиночный `=` — это присваивание ("положить значение в переменную"), а двойной `==` — сравнение ("равны ли значения"). Перепутать их — одна из самых частых ошибок у новичков в любом языке программирования, не только в Python.

Сравнивать можно и строки

Мы видели это в главе 8 — строки сравниваются по алфавиту символ за символом:

sravnenie_strok.py

```
print("apple" < "banana")   # True — "a" идёт раньше "b" в алфавите
print("Python" == "python") # False — регистр важен
```

Если это произошло — выполни команду!

Первый настоящий условный оператор: код, который выполняется только при определённых обстоятельствах.

Если это произошло — выполни команду!

Условный оператор `if` выполняет блок кода только тогда, когда условие после него истинно (`True`). Обратите внимание на двоеточие и отступ — они обязательны в Python:

if.py

```
age = 20

if age >= 18:
    print("Доступ разрешён.")
```

IndentationError

Строка после `if ...:` обязательно должна иметь отступ (обычно 4 пробела). Без отступа Python выдаст `IndentationError`. Отступ — не просто оформление: именно им Python определяет, какие строки относятся к блоку `if`, а какие уже нет.

А иначе?

`else` задаёт блок кода, который выполнится, если условие оказалось ложным:

if_else.py

```
age = 15

if age >= 18:
    print("Доступ разрешён.")
else:
    print("Доступ запрещён — вам ещё нет 18.")
```

Больше одного условия! :O

and, or и not объединяют несколько условий в одно.

Иногда решение зависит не от одного, а сразу от нескольких условий. Для этого есть три логических оператора: `and` (и), `or` (или), `not` (не).

logicheskie_operatory.py

```
age = 20
has_ticket = True

if age >= 18 and has_ticket:
    print("Проходите в зал.")

is_weekend = False
is_holiday = True
if is_weekend or is_holiday:
    print("Сегодня можно не ходить на
работу.")

if not has_ticket:
    print("Сначала нужно купить билет.")
```

Оператор	Истина, когда
`and`	оба условия истинны
`or`	хотя бы одно условие истинно
`not`	переворачивает значение на противоположное

Скобки помогают читать сложные условия

Когда условий много, скобки делают приоритет явным: `(age >= 18 and has_ticket) or is_vip` читается однозначно, в отличие от варианта без скобок.

Мини-проект — игра «Угадай число»

Первая настоящая игра книги — компьютер загадывает число, вы пытаетесь угадать.

Первая настоящая игра в этой книге! Компьютер загадывает число, вы вводите один вариант — и программа

говорит, угадали вы или нет (полноценный повтор попыток в цикле — уже в главе 10, здесь одна попытка).

ugadaj_chislo.py

```
import random

zagadannoe = random.randint(1, 20)
popytka = int(input("Угадайте число от 1 до 20: "))

if popytka == zagadannoe:
    print("Поздравляем, вы угадали!")
elif popytka < zagadannoe:
    print(f"Мимо! Загаданное число больше, чем {popytka}.")
else:
    print(f"Мимо! Загаданное число меньше, чем {popytka}.")

print(f"Загаданное число было: {zagadannoe}")
```

elif — «а иначе, если»

`elif` (сокращение от *else if*) добавляет дополнительное условие между `if` и `else`. Подробнее — в следующем разделе.

★★ Самостоятельная задача

Подсказка «горячо/холодно»

Добавьте третье условие: если разница между попыткой и загаданным числом меньше 3 — выведите «Очень близко!» перед основным сообщением.

Условия продолжают накапливаться!

elif и вложенные условия — когда одного if/else уже недостаточно.

Условий может быть сколько угодно — `elif` позволяет проверить их по очереди, одно за другим, пока одно из них не окажется истинным:

elif_cepochka.py

```
ocenka = 87

if ocenka >= 90:
    bukva = "A"
elif ocenka >= 80:
    bukva = "B"
elif ocenka >= 70:
    bukva = "C"
else:
    bukva = "D"

print(f"Оценка: {bukva}")
```

Порядок условий имеет значение

Python проверяет условия **по порядку** и останавливается на первом истинном — даже если следующие условия тоже подошли бы. Поэтому в примере выше условия идут от большего к меньшему: если бы `ocenka >= 70` стояло первым, оценка 87 неверно попала бы в категорию «C».

Вложенные условия

Внутри блока `if` может быть ещё один `if` — это называется **вложенным условием**:

vlozhennye_usloviya.py

```
age = 25
has_license = True

if age >= 18:
    if has_license:
        print("Можно водить машину.")
    else:
        print("Сначала получите права.")
else:
    print("Ещё рано водить машину.")
```

★★★ Задача повышенной сложности

Три вложенных уровня

Добавьте третий уровень: внутри «можно водить машину» проверьте ещё и наличие топлива в баке (третья переменная has_fuel).

Итоги

Что мы узнали в этой главе

- Тип `bool` хранит `True` или `False`; ноль и пустые значения — «ложь», всё остальное — «истина».
- Шесть операторов сравнения (`==` `!=` `<` `>` `<=` `>=`) превращают значения в bool.
- `if` выполняет блок кода при истинном условии; `else` — альтернативу, если условие ложно.
- `and`, `or`, `not` комбинируют несколько условий в одно.
- `elif` проверяет условия по очереди; условия можно вкладывать друг в друга.

Немного автоматизации!

Циклы for и while, break/continue — и наконец-то настоящая автоматизация всех фигур из глав 6-7.

Волшебные циклы!

Циклы for

Наконец-то настоящая автоматизация: вместо повторения кода вручную — просим Python повторить его самостоятельно.

Волшебные циклы!

Вспомните квадрат и шестиугольник из главы 6 — каждый состоял из одинаковых блоков команд, повторённых вручную. **Циклы** — это способ сказать Python «повтори это N раз» вместо того, чтобы копировать код руками.

Циклы for

Самый частый вид цикла в Python — `for`. Вместе с `range()` он умеет повторить блок кода заданное число раз:

cikl_for.py

```
for i in range(5):
    print(i)
```

`range(5)` генерирует числа от 0 до 4 (пять чисел, не включая 5) — `i` на каждом шаге принимает следующее из них.

range() с разными аргументами

`range(5)` — от 0 до 4. `range(2, 8)` — от 2 до 7. `range(0, 10, 2)` — от 0 до 8 с шагом 2: 0, 2, 4, 6, 8.

Когда переменная цикла не нужна

Если само число повторов важно, а не значение счётчика — по традиции его называют `_` (нижнее подчёркивание), сигнализируя «это значение сознательно не используется»:

Квадрат из главы 6: 8 строк → 2 строки

Без цикла (глава 6)

```
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
```

С циклом for

```
for _ in range(4):

artist.forward(100)
    artist.right(90)
```

Что использовать сегодня: цикл `for`. Он не «современнее» в смысле версии Python — циклы существуют с самых первых версий языка, — но именно ради него стоило дочитать до этой главы: то же самое поведение, в 4 раза короче, и легко изменить число повторов одной цифрой.

Условия if внутри циклов for

Циклы и условия отлично работают вместе — а циклы можно вкладывать друг в друга.

Условия if внутри циклов for

`if` из главы 9 прекрасно работает внутри цикла — на каждом шаге можно принимать своё решение:

if_v_cikle.py

```
for number in range(1, 11):
    if number % 2 == 0:
        print(number, "— чётное")
```

Вложенные циклы for

Цикл можно поместить внутрь другого цикла — тогда внутренний цикл выполняется полностью на каждом шаге внешнего:

vlozhennye_cikly.py

```
for row in range(3):
    for col in range(4):
        print(f"({row}, {col})", end=" ")
    print()
# новая строка после каждого ряда
```

Сколько раз выполнится внутренний цикл?

Внешний цикл выполняется 3 раза, внутренний — 4 раза *на каждом* шаге внешнего — итого `print(...)` внутри сработает `3 * 4 = 12` раз. Вложенные циклы — частая причина неожиданно долгого выполнения, если не следить за их количеством.

Перебор строк

for умеет перебирать не только числа — а while повторяет действие, пока условие остаётся истинным.

Перебор строк

Строка — это последовательность символов (глава 8), а значит, `for` умеет перебирать и её — символ за символом, без индексов:

perebor_strok.py

```
for letter in "Python":
    print(letter)
```

Циклы while

В отличие от `for`, который повторяется заданное число раз, `while` повторяется, **пока истинно условие** — сколько раз потребуется, заранее неизвестно:

cikl_while.py

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Бесконечный цикл

Если забыть `count += 1`, условие `count < 5` останется истинным навсегда — программа зависнет в **бесконечном цикле**. Всегда проверяйте, что внутри `while` есть шаг, который в итоге сделает условие ложным.

for или while?

Используйте `for`, когда число повторов известно заранее («нарисовать 4 стороны квадрата»). Используйте `while`, когда неизвестно («повторять, пока пользователь не угадает число»).

Прервать миссию! break и continue

Два способа управлять циклом изнутри: остановить его совсем или пропустить один шаг.

Иногда нужно выйти из цикла раньше времени или пропустить один шаг, не завершая цикл целиком — для этого есть два ключевых слова.

`break` — прервать цикл полностью

break.py

```
for number in range(1, 100):
    if number == 5:
        break  # цикл останавливается насовсем
    print(number)
```

Выведет только 1, 2, 3, 4 — как только `number` становится равным 5, цикл прерывается, не дожидаясь оставшихся 94 значений `range()`.

`continue` — пропустить этот шаг

continue.py

```
for number in range(1, 6):
    if number == 3:
        continue
# пропускаем оставшуюся часть тела цикла
для этого шага
    print(number)
```

Выведет 1, 2, 4, 5 — число 3 пропущено, но цикл продолжается дальше, в отличие от `break`.

break/continue работают и в while

Оба ключевых слова работают одинаково в цикле `while` — мы воспользуемся `break` в следующем разделе, переписывая игру «Угадай число».

Мини-проект — игра «Угадай число», версия 2

Та же игра, что и в главе 9, — но теперь с неограниченным числом попыток, благодаря while и break.

Помните игру «Угадай число» из главы 9? У неё была одна проблема — всего одна попытка. Теперь, вооружившись `while` и `break`, дадим игроку сколько угодно попыток.

ugadaj_chislo_v2.py

```
import random

zagadannoe = random.randint(1, 20)
popytki = 0

while True:
    popytka = int(input("Угадайте число
от 1 до 20: "))
    popytki += 1

    if popytka == zagadannoe:
        print(f"Поздравляем, вы угадали
за {popytki} попыток(ки)!")
        break
    elif popytka < zagadannoe:
        print("Загаданное число больше.")
    else:
        print("Загаданное число меньше.")
```

while True — намеренно бесконечный цикл

`while True:` создаёт цикл, который сам по себе никогда не остановится, — единственный выход из него: `break` внутри. Это распространённый и совершенно нормальный приём, когда условие остановки естественнее проверить в середине цикла, а не в его начале.

★★ Самостоятельная задача

Ограничение попыток

Добавьте ограничение — не более 5 попыток; если игрок не угадал за 5 попыток, сообщите правильный ответ и завершите игру.

Мини-проект — автоматизируем рисование квадрата

Возвращаемся к фигурам из главы 6 — на этот раз с циклами вместо ручного повторения.

Мини-проект — автоматизируем квадрат

Применим цикл к квадрату из главы 6 (мы уже видели этот пример в начале главы):

avto_kvadrat.py

```
for _ in range(4):
    artist.forward(100)
    artist.right(90)
```

Автоматизируем любую простую фигуру

Обобщим квадрат до функции-подобного шаблона, который рисует **любой** правильный многоугольник — вспомним формулу угла поворота из главы 6: `360 / количество_сторон`.

avto_lyubaya_figura.py

```
storony = 8        # хотим восьмиугольник
— просто меняем это число
dlina = 60
ugol = 360 / storony

for _ in range(storony):
    artist.forward(dlina)
    artist.right(ugol)
```

Одна переменная — любая фигура

Поменяйте `storony` на 3, 5, 10, 20 — программа автоматически нарисует треугольник, пятиугольник, десятиугольник или почти окружность, без единого изменения остального кода.

Мини-проект — автоматически рисуем мандалу

Та же мандала, что и в главе 6, — но короче и понятнее благодаря range() с тремя аргументами.

В главе 6 мандала рисовалась циклом `while` с ручным увеличением угла. Теперь, когда мы понимаем оба вида циклов, перепишем её на `for` с `range()` — станет ещё короче:

avto_mandala.py

```
shag_ugla = 10

for ugol in range(0, 360, shag_ugla):
    artist.setheading(ugol)
    artist.forward(150)
    artist.backward(150)
```

range() с тремя аргументами — то же, что и ручной while

`range(0, 360, shag_ugla)` генерирует именно те же числа, что мы получали вручную в главе 6: `while ugol < 360: ... ugol += shag_ugla`. Цикл `for` просто делает это короче и без риска забыть увеличить счётчик.

Мини-проект — спирали из дуг

Завершаем главу узором, который меняется на каждом шаге цикла, — и подводим итоги.

Финальный мини-проект главы: спираль из дуг — каждая следующая дуга немного больше предыдущей.

spirali_iz_dug.py

```
radius = 5

for _ in range(60):
    artist.circle(radius, 90)   # дуга в четверть окружности
    radius += 3                 # каждый раз чуть больше
```

Изменение переменной внутри цикла — обычное дело

В отличие от предыдущих примеров, здесь `radius` меняется *на каждом шаге* цикла — это и создаёт эффект нарастающей спирали, а не повторяющегося узора.

★★ Самостоятельная задача

Спираль из квадратов

Замените дугу на маленький квадрат (цикл на 4 стороны) внутри внешнего цикла — получится спираль из уменьшающихся или увеличивающихся квадратов.

Итоги

Что мы узнали в этой главе

- `for ... in range(n)` повторяет блок кода заданное число раз — и заменяет ручное копирование одинаковых строк.
- `for` умеет перебирать не только числа, но и символы строки.
- `while` повторяет действие, пока условие остаётся истинным — используется, когда число повторов заранее неизвестно.
- `break` прерывает цикл полностью; `continue`

пропускает текущий шаг и переходит к следующему.

- Циклы можно вкладывать друг в друга — тогда внутренний цикл выполняется полностью на каждом шаге внешнего.
- Все фигуры из глав 6–7, нарисованные вручную, теперь можно записать в 2–4 строки с циклом.

Очень много информации!

Списки, кортежи, множества и словари — четыре способа хранить сразу много данных.

Храним больше одного значения

Списки — самая универсальная коллекция Python: упорядоченный набор значений в одной переменной.

Храним больше одного значения

До сих пор каждая переменная хранила одно значение. Но что, если нужно хранить сразу список покупок, имена всех игроков или очки за каждый уровень? Для этого в Python есть четыре

встроенные **коллекции** — списки, кортежи, множества и словари. Начнём с самой универсальной — списков.

Списки

Список (`list`) — упорядоченная коллекция значений в квадратных скобках через запятую:

spiski.py

```
fruits = ["яблоко", "банан", "вишня"]
print(fruits)
print(type(fruits))
```

Список может хранить значения разных типов одновременно, хотя на практике чаще хранят значения одного вида:

smeshannyj_spisok.py

```
smeshannyj = ["Cartesian", 5, 3.14, True]
print(smeshannyj)
```

Доступ к значениям списка

Как и у строк (глава 8), у элементов списка есть индексы, начиная с нуля:

dostup_k_spisku.py

```
fruits = ["яблоко", "банан", "вишня"]
print(fruits[0])    # яблоко
print(fruits[-1])   # вишня — последний элемент
```

Делаем срез списка!

Срезы списков работают точно так же, как срезы строк, — уже знакомая логика в новом контексте.

Срезы списков работают точно так же, как срезы строк из главы 8:

srezy_spiskov.py

```
chisla = [10, 20, 30, 40, 50]
print(chisla[1:3])    # [20, 30]
print(chisla[:2])     # [10, 20]
print(chisla[2:])     # [30, 40, 50]
print(chisla[::-1])   # [50, 40, 30, 20, 10] — развёрнутый список
```

Срез списка — это новый список

Как и срез строки, срез списка возвращает **новый** список, не изменяя исходный.

Мощные операции со списками!

Списки изменяемы — у них есть целый набор методов для добавления, удаления, поиска и сортировки.

У списков (в отличие от строк) есть методы, которые **изменяют сам список** — списки, в отличие от строк, изменяемы.

Копирование и добавление

kopirovanie.py

```
original = [1, 2, 3]
kopiya = original.copy()
kopiya.append(4)
print(original)  # [1, 2, 3] — не изменился
print(kopiya)    # [1, 2, 3, 4]
```

kopiya = original — это не копия!

`kopiya = original` создаёт вторую переменную, указывающую **на тот же самый** список — изменение через одну переменную видно и через другую. Чтобы получить настоящую независимую копию, нужен `.copy()` (или срез `original[:]`).

Подсчёт и очистка

podschet.py

```
chisla = [1, 2, 2, 3, 2]
print(chisla.count(2))   # 3 — сколько раз встречается 2
chisla.clear()
print(chisla)             # [] — список пуст
```

Конкатенация

konkatenaciya_spiskov.py

```
a = [1, 2]
b = [3, 4]
print(a + b)   # [1, 2, 3, 4]
```

Поиск внутри списка

poisk.py

```
fruits = ["яблоко", "банан", "вишня"]
print("банан" in fruits)          # True
print(fruits.index("вишня"))      # 2 — индекс элемента
```

Добавление и удаление элементов

dobavlenie_udalenie.py

```
fruits = ["яблоко", "банан"]
fruits.append("вишня")        # добавить
в конец
fruits.insert(0, "манго")     # вставить
по индексу
print(fruits)

fruits.remove("банан")        # удалить
по значению
last = fruits.pop()            # удалить
и вернуть последний элемент
print(fruits, last)
```

Разворот и сортировка

razvorot_sortirovka.py

```
chisla = [3, 1, 4, 1, 5]
chisla.sort()
print(chisla)      # [1, 1, 3, 4, 5]
chisla.reverse()
print(chisla)      # [5, 4, 3, 1, 1]
```

Ещё больше интересного со списками!

Полезные встроенные функции, вложенные списки и более компактный способ строить новые списки.

Ещё несколько приёмов, которые часто пригождаются при работе со списками.

len(), min(), max(), sum()

vstroennye_funkcii.py

```
chisla = [4, 8, 15, 16, 23, 42]
print(len(chisla))   # 6 — количество элементов
print(min(chisla))   # 4
print(max(chisla))   # 42
print(sum(chisla))   # 108
```

Перебор списка циклом for

perebor_spiska.py

```
fruits = ["яблоко", "банан", "вишня"]
for fruit in fruits:
    print(fruit)
```

Списки списков (вложенные списки)

Элементом списка может быть другой список — так получаются таблицы (матрицы):

vlozhennye_spiski.py

```
matrix = [[1, 2, 3], [4, 5, 6]]
print(matrix[0])      # [1, 2, 3] — первая строка
print(matrix[0][1])   # 2 — второй элемент первой строки
```

Построение списка: цикл с append() → генератор списков

Классический подход

```
kvadraty = []
for n in range(1, 6):
    kvadraty.append(n ** 2)
print(kvadraty)
```

Современный Python (list comprehension)

```
kvadraty = [n ** 2 for n in range(1, 6)]
print(kvadraty)
```

Что использовать сегодня: list comprehension (генератор списков) для простых случаев — он существует в Python с версии 2.0, так что это не вопрос новизны, а вопрос стиля: короче и, при некоторой практике, читается как одно предложение («квадрат n для каждого n из диапазона»). Для более сложной логики (с несколькими условиями или побочными эффектами) цикл с `append()` часто остаётся понятнее — это не всегда "хуже", выбирайте то, что легче прочитать.

Мини-проект — автоматическая разноцветная звезда

Список цветов + цикл + Turtle — звезда, где каждый луч своего цвета.

Соединим списки с Turtle (главы 6–7): нарисуем звезду, где каждый луч — своего цвета из заранее заданного списка.

raznocvetnaya_zvezda.py

```
cveta = ["red", "orange", "yellow",
"green", "blue"]

for cvet in cveta:
    artist.pencolor(cvet)
    artist.forward(150)
    artist.right(144)  # угол
пятиконечной звезды из главы 6
```

Список любой длины — тот же код

Добавьте в `cveta` ещё пару значений — цикл `for cvet in cveta` сам подстроится под новую длину списка, без изменений остального кода.

★★ Самостоятельная задача

Случайные цвета

Замените список фиксированных цветов на random.choice() из списка — чтобы цвет каждого луча выбирался случайно.

Кортежи

Почти список, но неизменяемый — и именно поэтому иногда более подходящий выбор.

Кортеж (`tuple`) — почти то же самое, что список, но в круглых скобках и с одним принципиальным отличием: кортежи **неизменяемы** — после создания их нельзя изменить.

kortezhi.py

```
coords = (10, 20)
print(coords)
print(coords[0])   # 10 — индексация
работает как у списков
```

kortezh_nelzya_menyat.py

```
coords = (10, 20)
coords[0] = 99   # TypeError: 'tuple'
object does not support item assignment
```

Зачем нужны неизменяемые коллекции?

Неизменяемость — не недостаток, а гарантия: если вы передаёте координаты точки в другую часть программы как кортеж, вы точно знаете, что их никто случайно не изменит. Кортежи часто

используют для координат, RGB-цветов и других «пакетов» значений, которые логически представляют собой единое целое.

Распаковка кортежа

raspakovka.py

```
coords = (10, 20)
x, y = coords
print(x, y)   # 10 20
```

Мы уже пользовались распаковкой

`artist.position()` в модуле turtle возвращает именно такой кортеж `(x, y)` — можно сразу распаковать его в две переменные.

Множества

Коллекция без повторов — быстрый способ убрать дубликаты и сравнивать наборы значений.

Множество (`set`) — коллекция в фигурных скобках, у которой два ключевых отличия от списка: элементы в ней не повторяются, а порядок не гарантирован.

mnozhestva.py

```
chisla = {1, 2, 2, 3, 3, 3}
print(chisla)   # {1, 2, 3} — повторы исчезли сами
```

Зачем нужны множества?

Два самых частых случая: быстро убрать повторы из списка и быстро проверить, есть ли значение в коллекции (проверка через `in` у множества намного быстрее, чем у длинного списка).

primenenie_mnozhestv.py

```
spisok_s_povtorami = [1, 2, 2, 3, 1, 4]
unikalnye = set(spisok_s_povtorami)
print(unikalnye)   # {1, 2, 3, 4}
```

Операции над множествами

operacii_mnozhestv.py

```
a = {1, 2, 3}
b = {2, 3, 4}

print(a | b)   # объединение: {1, 2, 3,
4}
print(a & b)   # пересечение: {2, 3}
print(a - b)   # разность: {1}
```

Словари

Самая гибкая встроенная коллекция Python — доступ по ключу вместо числового индекса.

Словарь (`dict`) — самая гибкая коллекция: хранит пары «ключ — значение», а не просто значения по порядку. Вместо числового индекса доступ идёт по ключу — обычно строке.

slovari.py

```
student = {
    "name": "Cartesian",
    "age": 12,
    "city": "Москва",
}
print(student["name"])   # Cartesian
```

Добавление и изменение значений

izmenenie_slovarya.py

```
student["age"] = 13          # изменить существующее значение
student["grade"] = "7 класс" # добавить новый ключ
print(student)
```

Перебор словаря

perebor_slovarya.py

```
for key, value in student.items():
    print(f"{key}: {value}")
```

Проверка наличия ключа

proverka_klyucha.py

```
print("name" in student)      # True
print("phone" in student)     # False
```

KeyError — обращение к несуществующему ключу

`student["phone"]`, если такого ключа нет, вызовет `KeyError`. Более безопасный способ — метод `student.get("phone")`, который просто вернёт `None` вместо ошибки.

Мини-проект — бесконечные цвета

Словарь как «переводчик» — от слова к конкретному цвету Turtle.

Словарь отлично подходит для «перевода» одного значения в другое — например, названия цвета в его код. Нарисуем фигуру, «раскрашенную по словарю».

beskonechnye_cveta.py

```
cvetovaya_karta = {
    "огонь": "red",
    "трава": "green",
    "небо": "blue",
}

zapros = "небо"
artist.pencolor(cvetovaya_karta[zapros])
artist.circle(50)
```

★★ Самостоятельная задача

Добавьте свои цвета

Добавьте в cvetovaya_karta ещё 3-4 пары «слово — цвет» и нарисуйте для каждого отдельную фигуру.

Мини-проект — перестановка имени и фамилии

Собираем split(), распаковку и строки из главы 8 в одной короткой, но полезной программе — и подводим итоги.

Финальный мини-проект главы: разбить полное имя на части и собрать заново в другом порядке — «Фамилия Имя» вместо «Имя Фамилия», используя методы строк из главы 8 и списки из этой главы.

perestanovka_imeni.py

```
full_name = input("Введите имя и фамилию через пробел: ")
parts = full_name.split()   # split() возвращает список
name, surname = parts        # распаковка, как у кортежей

print(f"{surname} {name}")
```

split() возвращает список

Мы видели `.split()` в главе 8, но не заостряли внимание на том, что результат — это именно `list`. Теперь, зная про списки, можно распаковывать его так же, как распаковывали кортежи в разделе 11.6.

★★★ Задача повышенной сложности

Три части имени

Обработайте случай с отчеством (три слова): «Имя Отчество Фамилия» → «Фамилия И.О.» с инициалами.

Итоги

Что мы узнали в этой главе

- **Список** (`list`) — упорядоченная, изменяемая коллекция значений в `[]` .
- **Кортеж** (`tuple`) — как список, но неизменяемый, в `()` .
- **Множество** (`set`) — коллекция без повторов и без гарантированного порядка, в `{}` .
- **Словарь** (`dict`) — пары «ключ: значение», тоже в `{}` , но с двоеточиями.
- У списков десятки полезных методов: `append`, `insert`, `remove`, `pop`, `sort`, `reverse`, `index`, `count` и другие.

- Генератор списков `[выражение for элемент in коллекция]` — компактная альтернатива циклу с `append()`.

Множество увлекательных мини-проектов!

Шесть проектов, закрепляющих всё изученное в главах 1-11: условия, циклы, Turtle и списки.

Проект 12-1: Чётное или нечётное

Первый проект главы закрепляет условия и оператор % из ранних глав.

Эта глава — без новой теории: шесть проектов, которые заставляют работать вместе всё, что вы изучили в главах 1–11.

Часть 1 — Ваше число чётное или нечётное?

chetnoe_ili_nechetnoe.py

```
number = int(input("Введите число: "))

if number % 2 == 0:
    print(f"{number} — чётное.")
else:
    print(f"{number} — нечётное.")
```

Часть 2 — выводим чётные или нечётные числа из диапазона

chetnye_iz_diapazona.py

```
nachalo = int(input("Начало диапазона:
"))
konec = int(input("Конец диапазона: "))

chetnye = [n for n in range(nachalo,
konec + 1) if n % 2 == 0]
print("Чётные числа:", chetnye)
```

Какие темы здесь встретились

`input()` и `int()` — глава 8; `%` — глава 5; `if/else` — глава 9; генератор списков — глава 11.

Проект 12-2: Достаточно ли чаевых оставляет ваша мама?

Считаем процент чаевых от суммы счёта и оцениваем щедрость через цепочку elif.

Стандартные чаевые в ресторане — 15–20% от счёта. Проверим, укладывается ли конкретная сумма чаевых в этот диапазон.

chaevye.py

```
schet = float(input("Сумма счёта: "))
chaevye = float(input("Сумма чаевых: "))

procent = (chaevye / schet) * 100

if procent < 15:
    print(f"Маловато — всего
{procent:.1f}%. Обычно оставляют
15-20%.")
elif procent <= 20:
    print(f"В самый раз — {procent:.1f}
%!")
else:
    print(f"Очень щедро — целых
{procent:.1f}%!")
```

★★ Самостоятельная задача

Своя граница щедрости

Добавьте четвёртую категорию — «сказочно щедро» — для чаевых больше 30%.

Проект 12-3: Рисуем рождественскую ёлку

Несколько треугольных ярусов уменьшающегося размера, нарисованных циклом.

Нарисуем ёлку из треугольных «ярусов» уменьшающегося размера — используя цикл и фигуры из главы 6.

elka.py

```
artist.hideturtle()
artist.pencolor("green")
artist.fillcolor("green")

yarusy = 4
shirina = 120

artist.penup()
artist.goto(0, 100)
artist.pendown()

for yarus in range(yarusy):
    artist.begin_fill()
    artist.setheading(240)
    artist.forward(shirina)
    artist.setheading(0)
    artist.forward(shirina)
    artist.setheading(120)
    artist.forward(shirina)
    artist.end_fill()

    artist.penup()
    artist.setheading(270)
    artist.forward(30)
    artist.pendown()
    shirina -= 20
```

```
# ствол
artist.pencolor("brown")
artist.fillcolor("brown")
artist.setheading(270)
artist.penup()
artist.forward(10)
artist.pendown()
artist.begin_fill()
for _ in range(2):
    artist.forward(40)
    artist.right(90)
    artist.forward(20)
    artist.right(90)
artist.end_fill()
```

Каждый ярус — тот же треугольник, что и в главе 6

Формула угла поворота (`360 / 3 = 120`) — та же самая, что и для любого правильного многоугольника из главы 6. Ёлка — это просто несколько треугольников уменьшающегося размера, нарисованных друг под другом в цикле.

Проект 12-4: Спирали!

Один и тот же принцип — пять разных узоров, в зависимости от угла поворота.

Пять вариаций одной идеи — фигура, каждый шаг которой немного больше предыдущего.

Квадратная спираль

kvadratnaya_spiral.py

```
dlina = 5
for _ in range(60):
    artist.forward(dlina)
    artist.right(90)
    dlina += 3
```

Случайная спираль

sluchaynaya_spiral.py

```
import random

dlina = 5
for _ in range(60):
    artist.forward(dlina)
    artist.right(random.randint(80,
100))  # угол чуть-чуть «дрожит»
    dlina += 3
```

Треугольная спираль

treugolnaya_spiral.py

```
dlina = 5
for _ in range(60):
    artist.forward(dlina)
    artist.right(120)
    dlina += 3
```

Звёздная спираль

zvezdnaya_spiral.py

```
dlina = 5
for _ in range(100):
    artist.forward(dlina)
    artist.right(144)  # угол
пятиконечной звезды
    dlina += 2
```

Круговая спираль

krugovaya_spiral.py

```
radius = 5
for _ in range(60):
    artist.circle(radius, 90)
    radius += 3
```

Один общий принцип

Все пять спиралей — одна и та же идея из главы 10 («шаг + поворот + увеличение переменной») с разным углом поворота. Освоив один вариант, вы фактически освоили все пять.

Проект 12-5: Сложная мандала — полностью автоматизированная

Финальная версия мандалы из глав 6 и 10 — со случайными цветами и кругами на концах лучей.

Финальная эволюция мандалы из глав 6 и 10 — добавим случайный цвет на каждый луч и сделаем полностью автоматической, без единого «магического числа», прописанного вручную.

slozhnaya_mandala.py

```
import random

luchi = 36
shag_ugla = 360 / luchi
cveta = ["red", "orange", "purple",
"blue", "green"]

for i in range(luchi):
    artist.pencolor(random.choice(cveta))
    artist.setheading(i * shag_ugla)
    artist.forward(150)
    artist.circle(20)
    artist.forward(-150)
```

luchi определяет всё остальное

Измените только `luchi` — угол шага, число повторов цикла и даже плотность узора пересчитаются автоматически, ничего больше менять не нужно. Это и называется «полностью автоматизировано».

Проект 12-6: Гонка Turtle с использованием циклов

Несколько черепашек на одном экране одновременно — и подведение итогов главы.

Помните из главы 6, что экран и черепашка — разные объекты, и черепашек может быть несколько? Настало время это использовать: гонка из нескольких черепашек со случайным шагом.

gonka_turtle.py

```
import random

screen = turtle.Screen()
screen.setup(500, 400)

cveta = ["red", "blue", "green",
"orange"]
uchastniki = []

for i, cvet in enumerate(cveta):
    t = turtle.Turtle()
    t.shape("turtle")
    t.color(cvet)
    t.penup()
    t.goto(-200, i * 40 - 60)
    uchastniki.append(t)

finish_line = 200
pobeditel = None

while pobeditel is None:
    for t in uchastniki:
        t.forward(random.randint(1, 10))
        if t.xcor() >= finish_line:
            pobeditel = t.pencolor()
            break
```

```
print(f"Победила черепашка цвета
{pobeditel}!")
```

Список черепашек — тот же список из главы 11

`uchastniki` — обычный список Python (глава 11), просто хранящий не числа, а объекты `turtle.Turtle()`. Цикл `for t in uchastniki` перебирает его точно так же, как перебирал бы список чисел или строк.

★★★ Задача повышенной сложности

Финишная лента

Нарисуйте вертикальную линию на финише (x=200) перед началом гонки, используя отдельную черепашку-«судью».

Итоги

Что мы закрепили в этой главе

- Условия (`if/elif/else`) и операторы сравнения — на практике в проектах 12-1 и 12-2.
- Циклы (`for` , `while`) — основа всех фигур и спиралей в этой главе.
- Turtle: заливка, повороты, несколько черепашек на одном экране одновременно.
- Списки могут хранить любые объекты, включая другие объекты Python (например, черепашек), а не только числа и строки.

- Сложные визуальные эффекты (мандала, спирали, гонка) — это всегда комбинация нескольких простых, уже знакомых приёмов.

Автоматизация с помощью функций

Определение функций, аргументы, return, область видимости переменных и лямбда-функции.

Настоящая автоматизация

Функции переиспользуют набор действий столько раз, сколько нужно — без копирования кода.

Настоящая автоматизация

Циклы из главы 10 автоматизировали повторение одного и того же кода. Но что если один и тот же *набор действий* нужен в разных местах программы — не подряд, а время от времени? Копировать код каждый раз — плохая идея: любое

исправление придётся вносить во все копии. **Функции** решают эту проблему раз и навсегда.

Наша первая функция

pervaya_funkciya.py

```
def privetstvie():
    print("Привет, Python!")

privetstvie()
privetstvie()
privetstvie()
```

`def` **определяет** функцию — записывает её код, но пока не выполняет его. Функция выполняется только тогда, когда вы её **вызываете** — по имени со скобками: `privetstvie()`.

Определение — это ещё не вызов

Если написать только `def privetstvie(): ...` и не добавить `privetstvie()` ниже, программа не выведет ничего — Python просто запомнит функцию, но не запустит её.

Зачем нужны функции?

Аргументы позволяют одной и той же функции каждый раз делать что-то немного своё.

Зачем нужны функции?

- **Код не повторяется** — исправление вносится в одном месте.
- **Программу легче читать** — имя функции объясняет, что происходит, не заставляя вникать в детали реализации.
- **Легче искать ошибки** — если что-то сломалось в приветствии, вы точно знаете, где искать: внутри `privetstvie()`.

Каждый раз делаем что-то новое!

Функция без входных данных всегда делает одно и то же. Чтобы функция вела себя по-разному в зависимости от ситуации, ей передают **аргументы** — значения в скобках при вызове:

argumenty.py

```
def privetstvie(imya):
    print(f"Привет, {imya}!")

privetstvie("Ада")
privetstvie("Cartesian")
```

`imya` — **параметр** функции: имя, под которым переданное значение доступно внутри функции. При каждом вызове можно передать своё значение (**аргумент**).

Без аргументов?

Функция без параметров в скобках — например, `privetstvie()` из первого раздела — тоже совершенно нормальна: не каждой функции нужны входные данные.

Возвращаем ответ

return передаёт результат работы функции обратно туда, откуда она была вызвана.

До сих пор наши функции только печатали текст. Но часто нужно не вывести результат на экран, а **вернуть** его — чтобы использовать дальше в программе. Ключевое слово `return` делает именно это:

return.py

```
def summa(a, b):
    return a + b

result = summa(5, 7)
print(result)          # 12
print(summa(2, 3) * 10)  # результат функции можно использовать сразу — 50
```

print() внутри функции — это не то же самое, что return

Функция, которая только печатает результат, ничего не *возвращает* — попытка сохранить её результат в переменную даст `None`:

print_vs_return.py

```
def summa_pechataet(a, b):
    print(a + b)
# выводит на экран, но не возвращает

x = summa_pechataet(5, 7)  # на экране появится 12
print(x)                   # но x — это None!
```

return сразу завершает функцию

Как только выполняется `return`, функция немедленно заканчивает работу — код после него внутри функции не выполнится.

Нет аргументов? Слишком много аргументов!

Значения по умолчанию делают аргумент необязательным; *args принимает их сколько угодно.

Нет аргументов? Что делать!

Если вызвать функцию без обязательного аргумента, Python сообщит об ошибке. Чтобы аргумент можно было пропустить, ему задают **значение по умолчанию**:

znachenie_po_umolchaniyu.py

```
def privetstvie(imya="друг"):
    print(f"Привет, {imya}!")

privetstvie()          # Привет, друг!
privetstvie("Ада")     # Привет, Ада!
```

Слишком много аргументов!

Иногда заранее неизвестно, сколько аргументов понадобится передать. Символ `*` перед именем параметра собирает **любое** количество аргументов в один кортеж (глава 11):

args.py

```
def summa_vseh(*chisla):
    itog = 0
    for n in chisla:
        itog += n
    return itog

print(summa_vseh(1, 2))           # 3
print(summa_vseh(1, 2, 3, 4, 5))  # 15 — сколько угодно аргументов
```

Именованные аргументы

Аргументы можно передавать и по имени, а не только по порядку: `privetstvie(imya="Ада")` — это особенно удобно, когда у функции много параметров и легко перепутать порядок.

Глобальные и локальные переменные

Где «живёт» переменная и кто может её увидеть — важное правило, которое избавит от многих загадочных ошибок.

Переменные внутри функций

Переменная, созданная внутри функции, называется **локальной** — она существует только пока функция выполняется и недоступна снаружи:

lokalnye_peremennye.py

```
def moya_funkciya():
    message = "Я живу только внутри функции"
    print(message)

moya_funkciya()
print(message)   # NameError: message не определена здесь
```

Возвращаем локальные переменные

Чтобы значение, вычисленное внутри функции, стало доступно снаружи, его нужно вернуть через `return` — это единственный «официальный» способ передать что-то из функции наружу.

Глобальные переменные

Глобальная переменная объявлена вне всех функций — и доступна для чтения внутри любой из них:

globalnye_peremennye.py

```
ver = "1.0"   # глобальная переменная

def pokazat_versiyu():
    print(f"Версия программы: {ver}")
# чтение глобальной переменной — работает

pokazat_versiyu()
```

Изменить глобальную переменную изнутри функции — не так просто

Присваивание `ver = "2.0"` внутри функции создаст **новую локальную** переменную `ver`, а не изменит глобальную. Для настоящего изменения нужно ключевое слово `global` — но в реальном коде такое встречается редко: гораздо чище вернуть новое значение через `return`.

Лямбда-функции

Маленькие безымянные функции в одну строку — удобны в конкретных, узких случаях.

Лямбда-функция — это способ создать маленькую безымянную функцию в одну строку. Тот же результат, что и через `def`, но короче:

lambda.py

```
kvadrat = lambda x: x ** 2
print(kvadrat(5))   # 25
```

Слева от двоеточия — параметры (без скобок), справа — единственное выражение, результат которого автоматически возвращается (без явного `return`).

Где лямбда действительно полезна

Чаще всего лямбда-функции передают **как аргумент** в другую функцию — например, чтобы задать правило сортировки:

lambda_sortirovka.py

```
slova = ["python", "я",
"программирование"]
slova.sort(key=lambda word: len(word))
print(slova)   # ["я", "python",
"программирование"] — от короткого к
длинному
```

Простая функция: def → lambda

Обычная функция (def)

```
def kvadrat(x):
    return x ** 2

print(kvadrat(5))
```

Лямбда-функция

```
kvadrat =
lambda x: x ** 2

print(kvadrat(5))
```

Что использовать сегодня: обычную функцию с `def` — она читается яснее, у неё есть нормальное имя для отладки, и в неё легко добавить несколько строк логики или комментарий. `lambda` удобна только для одной короткой строки без имени — например, как аргумент функции `sorted()` или `max()`. Использовать lambda ради самой lambda, когда подошла бы обычная функция, — плохой стиль, а не «более современный».

Мини-проект — выполняем домашнее задание по математике с Python

Функции для генерации и проверки примеров на умножение — практика на все приёмы главы.

Соберём функции, аргументы и return в одном полезном инструменте: генераторе примеров для тренировки таблицы умножения.

domashnee_zadanie.py

```
import random

def sgenerirovat_primer():
    a = random.randint(2, 9)
    b = random.randint(2, 9)
    return a, b, a * b

def proverit_otvet(pravilnyj_otvet,
otvet_polzovatelya):
    return pravilnyj_otvet ==
otvet_polzovatelya

a, b, pravilnyj_otvet =
sgenerirovat_primer()
otvet = int(input(f"Сколько будет {a} x
{b}? "))

if proverit_otvet(pravilnyj_otvet,
otvet):
    print("Верно!")
else:
    print(f"Неверно — правильный ответ:
{pravilnyj_otvet}")
```

Функция возвращает сразу три значения

`return a, b, a * b` на самом деле возвращает один кортеж `(a, b, a * b)` — мы распаковываем его в три переменные сразу, как в главе 11.

★★ Самостоятельная задача

Счётчик правильных ответов

Оберните генерацию примера в цикл на 5 вопросов подряд и посчитайте, сколько из них решено верно.

Мини-проект — автоматизированные фигуры: новый уровень

Фигуры из главы 10 становятся настоящей переиспользуемой функцией — и подводим итоги.

Финальный проект главы: превратим повторяющийся код рисования фигур из главы 10 в настоящую функцию с аргументами.

figura_funkciya.py

```
def narisovat_figuru(storony, dlina):
    ugol = 360 / storony
    for _ in range(storony):
        artist.forward(dlina)
        artist.right(ugol)

narisovat_figuru(4, 100)    # квадрат
narisovat_figuru(3, 100)    #
треугольник, без повторения кода!
narisovat_figuru(8, 60)     #
восьмиугольник
```

Сравните с главой 10

В главе 10 для каждой фигуры пришлось бы менять переменные `storony` и `dlina` вручную перед каждым запуском. Теперь то же самое — параметры функции, и три фигуры рисуются тремя короткими строками подряд.

★★★ Задача повышенной сложности

Функция с позицией

Добавьте функции параметры x, y (со значениями по умолчанию 0, 0) — чтобы можно было указать, откуда начинать рисовать фигуру.

Итоги

Что мы узнали в этой главе

- `def имя(параметры):` определяет функцию; вызов `имя(аргументы)` её запускает.
- `return` передаёт результат функции наружу — это не то же самое, что `print()` внутри функции.
- Параметрам можно задать значение по умолчанию; `*args` принимает произвольное число аргументов.
- Переменные внутри функции — локальные; переменные вне всех функций — глобальные и доступны для чтения изнутри функций.

- `lambda` — короткая безымянная функция для простых случаев, чаще всего как аргумент другой функции.

Создаём объекты реального мира

Классы, объекты, атрибуты и методы — основы объектно-ориентированного программирования.

Что такое объектно-ориентированное программирование?

Вы уже пользовались объектами много глав подряд — просто ещё не называли их так.

Что такое объектно-ориентированное программирование?

Присмотритесь к коду из предыдущих глав: `artist.forward(100)`, `"Python".upper()`, `[1,2,3].append(4)` — во всех трёх у нас есть какой-то **объект** (черепашка, строка, список) и команда, которую мы у него вызываем через точку. Это и есть объектно-ориентированное программирование (ООП): способ организовать код вокруг объектов, у которых есть свои данные и свои действия — а не только вокруг отдельных функций и переменных.

Давайте это докажем!

Вы уже пользовались объектами с самой главы 6, даже не называя их так:

vy_uzhe_polzovalis_obektami.py

```
import turtle

artist = turtle.Turtle()   # artist — объект класса Turtle
artist.forward(100)
# forward() — действие (метод) этого объекта
artist.color("red")         # у объекта есть и своё состояние — например, цвет
```

`turtle.Turtle` — это **класс**: чертёж, шаблон, описывающий, какими бывают черепашки и что они умеют. Каждый вызов `turtle.Turtle()` создаёт новый, независимый **объект** (говорят также

«экземпляр класса») по этому чертежу — вспомните главу 12, где несколько независимых черепашек участвовали в гонке одновременно.

Классы

Пишем свой первый класс — чертёж, по которому Python будет создавать объекты.

Классы

Определим свой собственный класс — чертёж для объекта «Собака»:

klass_sobaka.py

```
class Sobaka:
    def __init__(self, klichka, vozrast):
        self.klichka = klichka
        self.vozrast = vozrast

rex = Sobaka("Рекс", 3)
print(rex.klichka, rex.vozrast)
```

`__init__` — специальный метод, который автоматически выполняется при создании объекта (`Sobaka("Рекс", 3)`) и настраивает его начальное состояние. `self` — ссылка на сам создаваемый объект; Python передаёт её автоматически, и первым параметром `__init__` (и любого другого метода) она должна быть всегда.

Класс vs объект — аналогия с формой для печенья

Класс — как форма для печенья: одна и та же форма может «вырезать» сколько угодно печений (объектов) — каждое своё, но по одному и тому же шаблону.

Объекты со своими значениями

Каждый объект хранит свои собственные значения атрибутов, независимо от других объектов того же класса:

raznye_obekty.py

```
rex = Sobaka("Рекс", 3)
sharik = Sobaka("Шарик", 5)

print(rex.klichka, rex.vozrast)      # Рекс 3
print(sharik.klichka, sharik.vozrast)  # Шарик 5 — независимо от rex
```

Управляем объектами

Атрибуты можно менять после создания объекта — а методы дают объекту собственные действия.

Управляем объектами

Значения атрибутов можно менять уже после создания объекта — так же, как менять значение обычной переменной:

menyaem_atributy.py

```
rex = Sobaka("Рекс", 3)
rex.vozrast = 4   # у Рекса был день рождения
print(rex.vozrast)
```

Объекты выполняют действия

Кроме данных (атрибутов), у класса могут быть свои действия — **методы**: обычные функции, определённые внутри класса, которые имеют доступ к `self` и, через него, к атрибутам объекта:

metody.py

```
class Sobaka:
    def __init__(self, klichka, vozrast):
        self.klichka = klichka
        self.vozrast = vozrast

    def layat(self):
        print(f"{self.klichka} говорит:
Гав-гав!")

    def prazdnovat_den_rozhdeniya(self):
        self.vozrast += 1
        print(f"Теперь {self.klichka}
исполнилось {self.vozrast}!")

rex = Sobaka("Рекс", 3)
rex.layat()
rex.prazdnovat_den_rozhdeniya()
```

Знакомая запись

`rex.layat()` устроена точно так же, как `artist.forward(100)` или `"текст".upper()` — методы объектов, которыми вы уже давно пользуетесь, устроены абсолютно так же, как метод `layat()`, который вы только что написали сами.

Гонка Turtle с объектами

Переписываем гонку из главы 12 с собственным классом — и подводим итоги главы.

Вернёмся к гонке черепашек из главы 12 — на этот раз обернём каждого участника в свой собственный класс, а не просто в объект `turtle.Turtle` напрямую.

gonka_s_klassom.py

```
import random

class Uchastnik:
    def __init__(self, cvet,
startovyj_y):
        self.t = turtle.Turtle()
        self.t.shape("turtle")
        self.t.color(cvet)
        self.t.penup()
        self.t.goto(-200, startovyj_y)
        self.cvet = cvet

    def sdelat_shag(self):
        self.t.forward(random.randint(1,
10))

    def finishiroval(self, finish_line):
        return self.t.xcor() >=
finish_line

cveta = ["red", "blue", "green",
"orange"]
uchastniki = [Uchastnik(cvet, i * 40 -
60) for i, cvet in enumerate(cveta)]

pobeditel = None
```

```
while pobeditel is None:
    for u in uchastniki:
        u.sdelat_shag()
        if u.finishiroval(200):
            pobeditel = u.cvet
            break

print(f"Победил участник цвета
{pobeditel}!")
```

Зачем оборачивать Turtle в свой класс?

В главе 12 логика гонки (движение, проверка финиша) была разбросана по основному коду. Теперь каждый участник — самостоятельный объект `Uchastnik`, который сам знает, как сделать шаг и как проверить, финишировал ли он. Для маленькой гонки разница не критична, но в более крупных программах именно так поддерживают порядок при росте кода.

★★★ Задача повышенной сложности

Счёт очков

Добавьте классу Uchastnik атрибут ochki и метод nabrat_ochki(n), увеличивающий счёт — начислите очки победителю в конце гонки.

Итоги

Что мы узнали в этой главе

- ООП организует код вокруг **объектов** — данных и действий, собранных вместе.
- `class` определяет чертёж объекта; `__init__` настраивает его начальное состояние.
- `self` — ссылка объекта на самого себя, обязательный первый параметр каждого метода.
- Атрибуты хранят данные объекта и независимы у каждого экземпляра класса.
- Методы — функции внутри класса, дающие объекту собственные действия —

работают точно так же, как уже знакомые `.forward()`, `.upper()`, `.append()`.

ГЛАВА 15 · СТР. 323

Python и файлы

Чтение и запись файлов на диске — данные, которые переживают завершение программы.

Зачем нужны файлы?

Переменные исчезают вместе с программой — файлы позволяют сохранить данные надолго.

Зачем нужны файлы?

Все переменные вашей программы исчезают, как только программа завершается. Чтобы сохранить данные — список результатов игры, текст,

настройки — между запусками программы, их нужно записать в **файл** на диске.

Открытие и чтение существующих файлов

chtenie_fajla.py

```
file = open("privet.txt", "r")   # "r" — режим чтения (read)
content = file.read()
print(content)
file.close()
# обязательно закрыть файл после работы
```

Не забывайте file.close()

Незакрытый файл может не сохранить изменения на диск или заблокировать доступ к нему для других программ. Забыть `close()` — очень частая ошибка новичков.

Современный способ: `with`

Ручное закрытие → менеджер контекста with

Классический подход

```
file = open("privet.txt", "r")
content = file.read()
print(content)
file.close()   # легко забыть!
```

Современный Python (with)

```
with open("privet.txt", "r") as file:
    content = file.read()
    print(content)
# файл закроется САМ, даже если внутри блока произойдёт ошибка
```

Что использовать сегодня: `with` — он существует в Python с версии 2.5, так что дело не в новизне, а в надёжности: файл закроется автоматически при выходе из блока `with`, даже если внутри что-то пойдёт не так. Начиная с этой главы в книге мы используем именно `with`.

Строка за строкой

Часто удобнее обработать файл по одной строке за раз, а не целиком.

Читать весь файл сразу одним куском не всегда удобно — особенно если файл большой. Часто нужнее обработать его **построчно**:

stroka_za_strokoj.py

```
with open("spisok.txt", "r") as file:
    for line in file:
        print(line.strip())   # strip()
убирает лишний перевод строки
```

Зачем нужен strip()

Каждая строка файла, кроме, возможно, последней, заканчивается невидимым символом перевода строки `\n`. `.strip()` (глава 8) убирает его, чтобы не было лишних пустых строк при выводе.

Все строки сразу — списком

readlines.py

```
with open("spisok.txt", "r") as file:
    lines = file.readlines()

print(len(lines), "строк")
print(lines[0])   # первая строка
```

Создание новых файлов

Записываем данные на диск — и знакомимся с современным способом работать с путями.

Создание новых файлов

Чтобы записать что-то в файл, откройте его в режиме записи `"w"` (write) — если файла не существует, Python создаст его сам; если существует — его прежнее содержимое будет стёрто:

zapis_fajla.py

```
with open("rezultaty.txt", "w") as file:
    file.write("Уровень 1: 100 очков\n")
    file.write("Уровень 2: 250 очков\n")
```

Работа с файлами

Три основных режима открытия файла:

- `"r"` — чтение (read); ошибка, если файла не существует.
- `"w"` — запись (write); создаёт новый файл или полностью стирает существующий.
- `"a"` — дозапись (append); добавляет текст в конец файла, не стирая то, что уже есть.

dozapis.py

```
with open("rezultaty.txt", "a") as file:
    file.write("Уровень 3: 400 очков\n")
```

Пути к файлам: строка или pathlib

Путь к файлу: обычная строка → pathlib

Классический подход

```
file_path = "data/rezultaty.txt"
with open(file_path, "r") as f:

print(f.read())
```

Современный Python (pathlib)

```
from pathlib import Path

file_path = Path("data") / "rezultaty.txt"
print(file_path.exists())
# удобные методы прямо у пути
with file_path.open("r") as f:
    print(f.read())
```

Что использовать сегодня: `pathlib` для путей к файлам — он появился в Python 3.4 и с тех пор считается предпочтительным способом. Строковые пути всё ещё повсеместно работают и встречаются в старом коде, но `pathlib` избавляет от ручной сборки пути через `+` и добавляет полезные методы вроде `.exists()` прямо у самого пути.

Мини-проект — знакомство с файлами

Дневник заметок, сохраняющийся между запусками программы, — и подведение итогов главы.

Соберём чтение и запись в одном небольшом проекте — дневнике заметок, который сохраняется между запусками программы.

dnevnik_zametok.py

```
from pathlib import Path

fajl_zametok = Path("zametki.txt")

novaya_zametka = input("Новая заметка: ")

with fajl_zametok.open("a") as f:
    f.write(novaya_zametka + "\n")

print("Все заметки:")
with fajl_zametok.open("r") as f:
    for line in f:
        print("-", line.strip())
```

Почему дозапись, а не перезапись

Режим `"a"` добавляет заметку, не стирая предыдущие — именно так и должен вести себя настоящий дневник: каждый запуск программы добавляет новую запись, а не начинает всё заново.

★★ Самостоятельная задача

Нумерация заметок

Добавьте номер к каждой заметке при чтении — например, «1. Первая заметка», «2. Вторая заметка».

Итоги

Что мы узнали в этой главе

- Файлы сохраняют данные между запусками программы — в отличие от обычных переменных.
- `with open(путь, режим) as file:` — современный и безопасный способ работать с файлами, закрывающий их автоматически.
- Режимы: `"r"` — чтение, `"w"` — запись с перезаписью, `"a"` — дозапись в конец.
- Файл можно читать целиком (`.read()`), построчно (цикл `for line in file`) или списком строк (`.readlines()`).

- `pathlib.Path` — современный способ работать с путями к файлам вместо голых строк.

ГЛАВА 16 · СТР. 335

Создаём классные приложения с Tkinter

Настоящие оконные приложения с кнопками, полями ввода и меню — модуль Tkinter.

Tkinter — правильно всё настраиваем!

Первое настоящее оконное приложение — с заголовком, размером и циклом событий.

До сих пор все программы либо выводили текст в терминал (главы 1–5, 8–15), либо рисовали на холсте Turtle (главы 6–7, 12). Пришло время настоящих **оконных приложений** с кнопками, полями ввода и меню — таких, как большинство привычных вам программ. Модуль для этого называется `Tkinter` и, как и `turtle`, входит в стандартную поставку Python.

pervoe_okno.py

```
import tkinter as tk

root = tk.Tk()
root.title("Моё первое приложение")
root.geometry("400x300")   # ширина x высота в пикселях

root.mainloop()   # запускает приложение и ждёт действий пользователя
```

`tk.Tk()` создаёт главное окно приложения — примерно как `turtle.Screen()` создавал холст для рисования. `root.mainloop()` запускает **цикл событий**: программа «замирает» в ожидании действий пользователя (клика, ввода текста) и реагирует на них — и продолжает работать, пока окно не будет закрыто.

root — распространённое соглашение об имени

Главное окно принято называть `root` («корень») — это не обязательное правило языка, а широко принятое соглашение, которое вы увидите почти в любом примере с Tkinter.

Метки, кнопки и их размещение

Первые интерактивные виджеты — и то, как Tkinter размещает их в окне.

Метки, кнопки и их размещение

Метка (`Label`) показывает текст, **кнопка** (`Button`) реагирует на клик. Оба виджета сначала создаются, а затем **размещаются** в окне — отдельным действием:

metki_knopki.py

```
import tkinter as tk

root = tk.Tk()

label = tk.Label(root, text="Привет,
Tkinter!")
label.pack()

def na_knopku_nazhali():
    print("Кнопку нажали!")

button = tk.Button(root, text="Нажми
меня", command=na_knopku_nazhali)
button.pack()

root.mainloop()
```

Параметр `command` связывает кнопку с функцией — важно передать саму функцию, **без скобок**: `command=na_knopku_nazhali`, а не `command=na_knopku_nazhali()`.

Частая ошибка: скобки в command

`command=na_knopku_nazhali()` вызовет функцию **немедленно**, один раз, при создании кнопки — а не при каждом клике. Кнопка свяжется с тем, что функция *вернёт*, а не с самой функцией.

Подробно о pack

`.pack()` — самый простой способ разместить виджет в окне. По умолчанию виджеты укладываются друг под другом сверху вниз, но есть полезные параметры:

pack_podrobno.py

```
label.pack(side="left", padx=10, pady=10)
# side: top (по умолчанию), bottom, left,
right
# padx/pady: внешний отступ по
горизонтали/вертикали
```

Множество полей ввода

Entry для одной строки, Text — для нескольких: два способа получить текст от пользователя.

Множество полей ввода

Поле ввода (`Entry`) — однострочное текстовое поле, куда пользователь может что-то напечатать:

polya_vvoda.py

```
import tkinter as tk

root = tk.Tk()

label = tk.Label(root, text="Введите
имя:")
label.pack()

entry = tk.Entry(root)
entry.pack()

def pokazat_imya():
    imya = entry.get()   # достаём текст
из поля
    print(f"Привет, {imya}!")

button = tk.Button(root,
text="Отправить", command=pokazat_imya)
button.pack()

root.mainloop()
```

`entry.get()` возвращает текущий текст поля — обычную строку, с которой можно работать, как с любой другой (глава 8).

Одна строка текста. Строка за строкой

`Entry` подходит для короткого текста в одну строку — имени, числа, пароля. Для многострочного текста используют другой виджет, `Text`:

mnogostrochnyj_tekst.py

```
text_box = tk.Text(root, height=5,
width=30)
text_box.pack()

def pokazat_tekst():
    content = text_box.get("1.0",
"end")   # с начала (строка 1, символ 0)
до конца
    print(content)
```

Индекс "1.0" — не опечатка

У `Text` своя система индексов: `"1.0"` означает «строка 1, символ 0» — то есть самое начало текста. Это не связано с числами `float` из главы 4 — просто текстовая метка позиции.

Переменные Tkinter

Специальные переменные, которые сами обновляют связанные с ними виджеты на экране.

Обычные переменные Python не «знают», когда их значение меняется в интерфейсе. Для виджетов, которые должны реагировать на изменения (переключатели, флажки — раздел 16.5), Tkinter предлагает свои специальные переменные: `StringVar`, `IntVar`, `BooleanVar`, `DoubleVar`.

peremennye_tkinter.py

```
import tkinter as tk

root = tk.Tk()

imya = tk.StringVar(value="Гость")

label = tk.Label(root,
textvariable=imya)   # метка сама
обновится при смене imya
label.pack()

def smenit_imya():
    imya.set("Cartesian")

button = tk.Button(root, text="Сменить
имя", command=smenit_imya)
button.pack()

root.mainloop()
```

Значение достают методом `.get()` и меняют методом `.set()` — а виджеты, связанные через `textvariable`, обновляются на экране автоматически.

Множество вариантов!

Переключатели для выбора одного варианта и флажки для независимых да/нет решений.

Для выбора из нескольких вариантов у Tkinter есть переключатели (`Radiobutton` — выбрать один из нескольких) и флажки (`Checkbutton` — включить/выключить каждый независимо).

radiobutton.py

```
import tkinter as tk

root = tk.Tk()
vybor = tk.StringVar(value="chay")

tk.Radiobutton(root, text="Чай",
variable=vybor, value="chay").pack()
tk.Radiobutton(root, text="Кофе",
variable=vybor, value="kofe").pack()

def pokazat_vybor():
    print(f"Вы выбрали: {vybor.get()}")

tk.Button(root, text="Готово",
command=pokazat_vybor).pack()
root.mainloop()
```

checkbutton.py

```
saharok = tk.BooleanVar(value=False)
tk.Checkbutton(root, text="С сахаром",
variable=saharok).pack()

def pokazat_flazhok():
    print(f"С сахаром: {saharok.get()}")
```

Меню

Настоящее приложение почти всегда начинается с меню в верхней части окна.

Настоящее приложение редко обходится без меню в верхней части окна. В Tkinter меню строится из объекта `Menu`:

menu.py

```
import tkinter as tk

root = tk.Tk()

def novyj_fajl():
    print("Создаём новый файл...")

menu_bar = tk.Menu(root)
fajl_menu = tk.Menu(menu_bar, tearoff=0)
fajl_menu.add_command(label="Новый",
command=novyj_fajl)
fajl_menu.add_separator()
fajl_menu.add_command(label="Выход",
command=root.quit)
menu_bar.add_cascade(label="Файл",
menu=fajl_menu)

root.config(menu=menu_bar)
root.mainloop()
```

tearoff=0

По умолчанию Tkinter добавляет в начало каждого меню пунктирную линию, позволяющую «оторвать» меню в отдельное окно — устаревшая функция, которую в современных интерфейсах почти всегда отключают параметром `tearoff=0`.

Идеальная компоновка — grid

Для форм с несколькими колонками grid() удобнее, чем pack().

`.pack()` прекрасно подходит для простых случаев, но у него есть предел: сложные формы с несколькими колонками собрать сложно. Для этого есть `.grid()` — размещение по строкам и столбцам, как в таблице:

grid.py

```
import tkinter as tk

root = tk.Tk()

tk.Label(root, text="Имя:").grid(row=0,
column=0)
tk.Entry(root).grid(row=0, column=1)

tk.Label(root,
text="Email:").grid(row=1, column=0)
tk.Entry(root).grid(row=1, column=1)

tk.Button(root,
text="Отправить").grid(row=2, column=0,
columnspan=2)

root.mainloop()
```

Не смешивайте pack() и grid() в одном контейнере

Виджеты внутри одного и того же родительского окна (или фрейма) должны использовать либо только `pack()`, либо только `grid()` — смешение вызывает ошибку или зависание интерфейса. Разные контейнеры (например, вложенные `Frame`) могут использовать разные способы размещения независимо друг от друга.

Мини-проект — приложение-калькулятор чаевых

Первое полноценное приложение книги — с полями ввода, кнопкой и результатом на экране.

Соберём всё изученное в главе в одном настоящем приложении: калькулятор чаевых с полем ввода, кнопкой и меткой результата.

kalkulyator_chaevyh.py

```
import tkinter as tk

root = tk.Tk()
root.title("Калькулятор чаевых")

tk.Label(root, text="Сумма
счёта:").grid(row=0, column=0, padx=10,
pady=10)
schet_entry = tk.Entry(root)
schet_entry.grid(row=0, column=1)

tk.Label(root, text="Процент
чаевых:").grid(row=1, column=0, padx=10,
pady=10)
procent_entry = tk.Entry(root)
procent_entry.grid(row=1, column=1)

rezultat_label = tk.Label(root, text="",
font=("Arial", 14, "bold"))
rezultat_label.grid(row=3, column=0,
columnspan=2, pady=10)

def poschitat():
    schet = float(schet_entry.get())
    procent = float(procent_entry.get())
    chaevye = schet * procent / 100
```

```
    rezultat_label.config(text=f"Чаевые:
{chaevye:.2f}")

tk.Button(root, text="Посчитать",
command=poschitat).grid(row=2, column=0,
columnspan=2)

root.mainloop()
```

Какие темы здесь встретились

`grid()` — этот раздел; `float()` — глава 4; форматирование `:.2f` — глава 8; сама формула чаевых — глава 12, проект 12-2.

★★★ Задача повышенной сложности

Проверка ввода

Оберните вычисление в try/except (забегая немного вперёд — подробно об этом в главе 21) — чтобы приложение не падало, если пользователь введёт не число, а текст.

Итоги

Что мы узнали в этой главе

- `tk.Tk()` создаёт главное окно; `mainloop()` запускает цикл обработки событий.
- Основные виджеты: `Label`, `Button`, `Entry`, `Text`, `Radiobutton`, `Checkbutton`.
- `command=функция` связывает кнопку с действием — без скобок после имени функции.
- `pack()` и `grid()` — два способа размещения виджетов; не смешивать их в одном контейнере.
- Специальные переменные Tkinter (`StringVar` и другие)

автоматически обновляют связанные с ними виджеты.

Проект: игра «Крестики-нолики» с Tkinter

Первая полноценная игра книги — собираем её шаг за шагом, от привязки событий до готовой программы.

Привязка событий — делаем приложения динамическими!

command — не единственный способ реагировать на действия пользователя: bind() открывает события любого рода.

В главе 16 кнопки реагировали на клик через параметр `command` — это самый частый, но не единственный способ связать код с действием пользователя. **Привязка событий** (`.bind()`) позволяет реагировать на что угодно: нажатие клавиши, движение мыши, наведение курсора.

privyazka_sobytij.py

```
import tkinter as tk

root = tk.Tk()
label = tk.Label(root, text="Нажмите
любую клавишу", font=("Arial", 14))
label.pack(padx=20, pady=20)

def na_klavishu(event):
    label.config(text=f"Вы нажали:
{event.keysym}")

root.bind("<Key>", na_klavishu)
root.mainloop()
```

Функция-обработчик события всегда принимает один параметр — `event` — объект с подробностями о произошедшем событии: какая клавиша нажата (`event.keysym`), где кликнула мышь, и так далее.

Пригодится в главе 19

Именно `.bind()` позволит змейке в главе 19 реагировать на нажатия клавиш со стрелками — тот же самый приём, только с другим типом события.

Игра «Крестики-нолики» — объяснение

Разберём план из шести шагов, по которому мы соберём игру — от пустого окна до готовой программы.

Игра «Крестики-нолики» — объяснение

Правила знакомы почти всем: поле 3×3, два игрока по очереди ставят «X» и «O», первый, выстроивший три своих символа подряд — по горизонтали, вертикали или диагонали — побеждает. Соберём эту игру шаг за шагом — так, как её строил бы разработчик, а не одним куском кода сразу:

1. Настроить окно;
2. Создать переменные состояния игры;
3. Создать поле из девяти кнопок;
4. Реагировать на клик — рисовать X или O;

5. Проверять после каждого хода, есть ли победитель;
6. Добавить кнопку «Новая игра».

Настраиваем Tkinter

nastrojka_okna.py

```
import tkinter as tk

root = tk.Tk()
root.title("Крестики-нолики")
```

Создаём глобальные переменные

Игре нужна память между ходами — заведём переменные состояния и построим поле из кнопок циклом.

Создаём глобальные переменные

Игре нужно помнить своё состояние между ходами — чей сейчас ход, закончена ли игра, и сами кнопки поля, чтобы менять их текст:

globalnye_peremennye.py

```
tekuschij_igrok = "X"
polya = []              # список из 9
кнопок поля
igra_okonchena = False
```

Создаём кнопки

Поле 3×3 — это девять кнопок, расположенных через `grid()` (глава 16). Создадим их циклом, а не девятью одинаковыми строками кода вручную (глава 10):

sozdaem_knopki.py

```
pole_frame = tk.Frame(root)
pole_frame.grid(row=1, column=0,
columnspan=3)

for indeks in range(9):
    knopka = tk.Button(
        pole_frame,
        text="",
        font=("Arial", 24, "bold"),
        width=3, height=1,
    )
    knopka.grid(row=indeks // 3,
column=indeks % 3)
    polya.append(knopka)
```

indeks // 3 и indeks % 3 — координаты из номера

Числа от 0 до 8 нужно превратить в строку и столбец таблицы 3×3. Целочисленное деление `//` (глава 5) даёт номер строки (`0,0,0,1,1,1,2,2,2`), а остаток `%` — номер столбца (`0,1,2,0,1,2,0,1,2`).

При нажатии на кнопку рисуем на ней

Каждая кнопка должна знать свой номер — здесь прячется одна из самых известных тонкостей Python.

Каждой кнопке нужна своя функция-обработчик клика, которая знает *какая именно* кнопка нажата. Хитрость в том, чтобы связать `command` с номером кнопки — для этого используем лямбда-функцию из главы 13:

risuem_na_knopke.py

```
def na_knopku_nazhali(indeks):
    global tekuschij_igrok

    if polya[indeks]["text"] != "":
        return   # клетка уже занята

    polya[indeks]["text"] =
tekuschij_igrok
    tekuschij_igrok = "O" if
tekuschij_igrok == "X" else "X"

# при создании каждой кнопки:
knopka.config(command=lambda i=indeks:
na_knopku_nazhali(i))
```

Зачем lambda i=indeks, а не просто lambda: na_knopku_nazhali(indeks)?

Без `i=indeks` все девять кнопок запомнили бы одну и ту же переменную `indeks` — а не её значение в момент создания. К моменту клика цикл давно завершится, и `indeks` будет равен 8 для всех кнопок сразу. Значение по умолчанию `i=indeks` «замораживает» текущее значение при создании каждой лямбды — классическая и важная тонкость Python.

Проверяем после каждого хода, победил ли игрок

Восемь возможных выигрышных линий — три строки, три столбца, две диагонали.

После каждого хода нужно проверить: не выстроились ли три одинаковых символа подряд. Всего в игре восемь возможных «выигрышных линий» — три строки, три столбца и две диагонали:

proverka_pobedy.py

```
def proverit_pobedu():
    linii_pobedy = [
        (0, 1, 2), (3, 4, 5), (6, 7,
8),  # строки
        (0, 3, 6), (1, 4, 7), (2, 5,
8),  # столбцы
        (0, 4, 8), (2, 4,
6),             # диагонали
    ]
    for a, b, c in linii_pobedy:
        znacheniya = (polya[a]["text"],
polya[b]["text"], polya[c]["text"])
        if znacheniya[0] != "" and
znacheniya[0] == znacheniya[1] ==
znacheniya[2]:
            return znacheniya[0]   # 'X'
или '0' — есть победитель
    return None   # победителя пока нет
```

Список кортежей — компактный способ описать 8 линий

Каждая линия — кортеж (глава 11) из трёх индексов поля. Список из восьми таких кортежей заменяет восемь отдельных проверок `if`, которые пришлось бы писать вручную.

Ничья

Если все девять клеток заполнены, а победителя нет — это ничья:

proverka_nichyej.py

```
def polye_zapolneno():
    return all(knopka["text"] != "" for
knopka in polya)
```

Кнопка новой игры

Финальный штрих — сброс игры без перезапуска программы — и полная работающая программа целиком.

Кнопка новой игры

Чтобы не перезапускать программу заново после каждой партии, добавим кнопку сброса — она возвращает все переменные и все клетки в начальное состояние:

novaya_igra.py

```
def novaya_igra():
    global tekuschij_igrok,
igra_okonchena
    tekuschij_igrok = "X"
    igra_okonchena = False
    for knopka in polya:
        knopka.config(text="")
```

Полная программа

Соберём все части в одну программу — она уже полностью проверена и доступна отдельным файлом в этой книге:

projects/tkinter/tic-tac-toe/tic_tac_toe.py

Запустите игру у себя

Скачайте файл и запустите его через `python tic_tac_toe.py` в терминале (глава 3) — либо кнопкой Run в VS Code или PyCharm.

★★ Самостоятельная задача

Счёт побед

Добавьте метки счёта побед X и O, увеличивайте нужный счётчик при каждой победе — счётчик не должен обнуляться кнопкой «Новая игра».

★★★ Задача повышенной сложности

Подсветка выигрышной линии

Когда находится победитель, измените цвет фона трёх выигрышных кнопок — потребуется вернуть из proverit_pobedu() не только победителя, но и индексы линии.

Итоги

Что мы узнали в этой главе

- `.bind()` реагирует на любые события — не только клик по кнопке.
- Большие проекты строят по шагам: состояние → интерфейс → обработка событий → правила → сброс.
- Лямбда с параметром по умолчанию (`lambda i=indeks: ...`) «замораживает» значение переменной цикла в момент создания.
- Список кортежей — удобный способ описать несколько похожих проверок (восемь

выигрышных линий) без повторения кода.

- У полноценной игры почти всегда есть способ начать заново, не перезапуская программу.

Проект: приложение для рисования с Tkinter

Собственная «рисовалка» с холстом, выбором фигур, цвета и толщины линии.

Приложение для рисования — объяснение

Разберём план из шести шагов — от пустого холста до готовой рисовалки.

Приложение для рисования — объяснение

Соберём собственный маленький «Paint»: холст, на котором можно рисовать линии, прямоугольники, овалы

и произвольные каракули мышью — с выбором цвета и толщины. План такой же пошаговый, как и в главе 17:

1. Холст для рисования (виджет `Canvas`);
2. Панель инструментов — кнопки выбора фигуры;
3. Реакция на движение и клики мыши;
4. Собственно рисование линий, прямоугольников, овалов;
5. Выбор толщины и цвета;
6. Кнопка очистки холста.

Начинаем работу

nachalo.py

```
import tkinter as tk

root = tk.Tk()
root.title("Рисовалка")
```

Настраиваем экран. Создаём холст

Виджет Canvas — прямоугольная область для рисования внутри окна Tkinter.

Настраиваем экран

nastrojka_ekrana.py

```
root.title("Рисовалка")
```

Создаём холст

Виджет `Canvas` — прямоугольная область, на которой можно рисовать линии, фигуры и текст координатами, как в Turtle (главы 6–7), только внутри окна Tkinter:

sozdaem_holst.py

```
canvas = tk.Canvas(root, width=600,
height=400, bg="white")
canvas.pack()
```

Координаты Canvas — как у экрана, не как у Turtle

У `Canvas` точка (0, 0) — левый верхний угол, а не центр, как у экрана Turtle, и ось Y растёт **вниз**, а не вверх. Если раньше вы рисовали Turtle — обратите на это внимание, чтобы не запутаться.

Создаём первое меню (фигуры)

Панель инструментов с кнопками выбора фигуры — и переменные, хранящие текущее состояние.

Создаём первое меню (фигуры)

Панель инструментов — обычный `Frame` (глава 16) с кнопками, каждая из которых выбирает свою фигуру:

menu_figur.py

```
toolbar = tk.Frame(root)
toolbar.pack(side="top", fill="x")

def vybrat_figuru(figura):
    global tekuschaya_figura
    tekuschaya_figura = figura

tk.Button(toolbar, text="Линия",
command=lambda:
vybrat_figuru("linia")).pack(side="left")
tk.Button(toolbar, text="Прямоугольник",
command=lambda:
vybrat_figuru("pryamougolnik")).pack(side="l
tk.Button(toolbar, text="Овал",
command=lambda:
vybrat_figuru("oval")).pack(side="left")
```

Заставляем параметры рисования работать!

Заведём глобальные переменные для текущей фигуры, цвета и толщины линии — их будут менять кнопки панели инструментов и читать функции рисования:

parametry.py

```
tekuschaya_figura = "linia"
tekuschij_cvet = "black"
tolschina = 3
```

Получаем позицию мыши

События мыши позволяют превратить движение курсора в настоящую линию на холсте.

Получаем позицию мыши

Как и в главе 17, реагировать на мышь помогает `.bind()` — только вместо клавиатурных событий здесь события мыши: `<Button-1>` (нажатие левой кнопки), `<B1-Motion>` (движение с зажатой левой кнопкой), `<ButtonRelease-1>` (кнопку отпустили).

```
poziciya_myshi.py
```

```
def pokazat_poziciyu(event):

pozicia_label.config(text=f"x={event.x},
y={event.y}")

canvas.bind("<Motion>", pokazat_poziciyu)
```

`event.x` и `event.y` — координаты курсора относительно холста, в тех же единицах, что и у самого `Canvas`.

Рисуем линии

Три события вместе создают эффект «рисования от точки до точки»: запоминаем начало на `<Button-1>`, рисуем линию к текущей позиции на каждом `<B1-Motion>`:

risuem_linii.py

```
start_x, start_y = None, None

def nachalo_risovaniya(event):
    global start_x, start_y
    start_x, start_y = event.x, event.y

def vo_vremya_risovaniya(event):
    canvas.create_line(start_x, start_y,
event.x, event.y, fill=tekuschij_cvet,
width=tolschina)

canvas.bind("<Button-1>",
nachalo_risovaniya)
canvas.bind("<B1-Motion>",
vo_vremya_risovaniya)
```

create_line — как forward() у Turtle, только по координатам

`canvas.create_line(x1, y1, x2, y2)` рисует прямую между двумя точками — концептуально то же самое, что `goto()` у Turtle из главы 6, только без «черепашки», которая сама туда едет.

Квадраты и прямоугольники! Круги и овалы!

Canvas умеет рисовать готовые фигуры по двум углам — не только линии.

Прямоугольники и овалы рисуются похожим образом — Canvas умеет строить их сразу по двум противоположным углам, без ручного вычисления сторон:

pryamougolniki_ovaly.py

```
def vo_vremya_risovaniya(event):
    if tekuschaya_figura ==
"pryamougolnik":
        canvas.create_rectangle(
            start_x, start_y, event.x,
event.y,
            outline=tekuschij_cvet,
width=tolschina,
        )
    elif tekuschaya_figura == "oval":
        canvas.create_oval(
            start_x, start_y, event.x,
event.y,
            outline=tekuschij_cvet,
width=tolschina,
        )
```

Промежуточные фигуры множатся

Если рисовать так, как показано выше, при каждом движении мыши будет создаваться **новая** фигура поверх предыдущей — а не одна, растущая вместе с движением. В полной программе (раздел 18.7) эта проблема решена: предыдущая «черновая» фигура удаляется перед тем, как нарисовать новую.

Выбираем размер!

Очень много цветов!

Ползунок толщины и палитра цветов, построенная циклом по списку.

Выбираем размер!

Виджет `Scale` — ползунок для выбора числа в диапазоне, отлично подходит для толщины линии:

vybor_razmera.py

```
def vybrat_tolschinu(znachenie):
    global tolschina
    tolschina = int(znachenie)

tolschina_scale = tk.Scale(toolbar,
from_=1, to=10, orient="horizontal",
command=vybrat_tolschinu)
tolschina_scale.set(3)
tolschina_scale.pack(side="left")
```

command у Scale получает значение, а не событие

В отличие от `.bind()`, обработчик `Scale` получает готовое строковое значение ползунка напрямую — поэтому `vybrat_tolschinu(znachenie)` сразу превращает его в число через `int()` (глава 4).

Очень много цветов!

Палитру цветов легко построить циклом (глава 10) по списку названий (глава 11) — вместо того, чтобы создавать каждую кнопку вручную:

vybor_cveta.py

```
cveta = ["black", "red", "blue",
"green", "orange", "purple"]
for cvet in cveta:
    tk.Button(toolbar, bg=cvet, width=2,
command=lambda c=cvet:
vybrat_cvet(c)).pack(side="left")
```

lambda c=cvet — та же тонкость, что и в главе 17

Без `c=cvet` все кнопки палитры выбирали бы **последний** цвет из списка — та же самая ловушка с лямбдами внутри цикла, что мы уже разбирали в проекте «Крестики-нолики».

Я закончил рисовать! Полная программа

Последние штрихи — очистка холста и свободное рисование — и полная работающая программа целиком.

Я закончил рисовать!

Осталось добавить кнопку очистки холста и режим «свободного рисования» (карандаш) — для него точки рисуются на каждое движение мыши без привязки к начальной точке:

ochistka_i_svobodno.py

```
def ochistit_holst():
    canvas.delete("all")

def vo_vremya_risovaniya(event):
    if tekuschaya_figura == "svobodno":
        canvas.create_oval(
            event.x - tolschina, event.y
- tolschina,
            event.x + tolschina, event.y
+ tolschina,
            fill=tekuschij_cvet,
outline=tekuschij_cvet,
        )
        return

# ... остальные фигуры, как в разделе
18.5
```

Полная программа

Полная версия приложения, включая исправленную отрисовку «черновых» фигур во время перетаскивания мыши, доступна отдельным файлом:

projects/tkinter/paint-app/paint_app.py

★★ Самостоятельная задача

Кнопка «Отменить»

Сохраняйте id каждой нарисованной фигуры (canvas.create_* возвращает id) в список — и добавьте кнопку, удаляющую последнюю через canvas.delete(id).

★★★ Задача повышенной сложности

Сохранение рисунка

Изучите модуль tkinter.filedialog и попробуйте добавить кнопку «Сохранить» — как минимум сохраняющую список нарисованных фигур в текстовый файл (глава 15).

Итоги

Что мы узнали в этой главе

- `Canvas` — виджет Tkinter для рисования линий и фигур по координатам; координаты растут вправо и вниз от левого верхнего угла.
- `canvas.create_line/rectangle/oval(...)` рисуют готовые фигуры по координатам.
- События мыши (`<Button-1>`, `<B1-Motion>`, `<ButtonRelease-1>`) отслеживают клик, перетаскивание и отпускание кнопки мыши.
- `Scale` — ползунок для выбора числа в диапазоне.

- Палитра из нескольких похожих кнопок эффективнее строится циклом, чем вручную одна за другой.

Проект: игра «Змейка» с Turtle

Классическая игра целиком на Turtle — движение, еда, счёт и столкновения.

Игра «Змейка»

Классическая игра — движение, еда и столкновения — на уже знакомом модуле turtle.

Игра «Змейка»

«Змейка» — одна из самых узнаваемых игр в истории программирования. Змейка непрерывно движется по полю, съедает появляющиеся яблоки и растёт с каждым съеденным яблоком; игра заканчивается при столкновении со стеной или с собственным хвостом. Соберём её на уже знакомом модуле

`turtle` (главы 6–7, 12) — только на этот раз черепашка станет не художником, а персонажем игры.

Импортируем необходимые модули

importy.py

```
import random
import turtle
```

`random` понадобится для случайного положения яблока (глава 5), `turtle` — для самой графики.

Настраиваем экран Turtle

Отключаем автообновление экрана и заводим переменные, которые будут помнить состояние игры.

Настраиваем экран Turtle

Игре нужен предсказуемый, контролируемый экран — и, что важно, отключённое автообновление: обновлять картинку мы будем вручную, ровно раз за игровой шаг, а не при каждом мелком движении:

nastrojka_ekrana.py

```
screen = turtle.Screen()
screen.title("Змейка")
screen.bgcolor("black")
screen.setup(width=600, height=600)
screen.tracer(0)   # отключаем
автообновление
```

Зачем tracer(0)?

Без `tracer(0)` Turtle перерисовывает экран после *каждого* отдельного движения — для игры, где нужно двигать голову и несколько сегментов тела на каждом шаге, это выглядело бы мерцающим и медленным. `tracer(0)` + ручной `screen.update()` в конце каждого шага дают куда более плавную анимацию.

Создаём и инициализируем необходимые переменные

peremennye.py

```
RAZMER_SHAGA = 20
GRANICA = 280

napravlenie = "stop"
schet = 0
igra_okonchena = False
segmenty = []   # тело змейки
```

ЗАГЛАВНЫЕ_БУКВЫ — соглашение для констант

`RAZMER_SHAGA` и `GRANICA` не меняются в процессе игры — по соглашению такие «постоянные» переменные принято называть заглавными буквами. Само по себе это ничего не меняет технически, но сигнализирует читателю: «это значение не должно меняться».

Рисуем голову

Голова и яблоко — обычные объекты Turtle с разной формой и цветом.

Рисуем голову

Голова змейки — обычный объект `turtle.Turtle()`, знакомый ещё с главы 6, только с квадратной формой и без рисования линий (перо поднято):

golova.py

```
golova = turtle.Turtle()
golova.speed(0)
golova.shape("square")
golova.color("white")
golova.penup()
golova.goto(0, 0)
```

Рисуем первое яблоко

Яблоко — тоже черепашка, только круглая и красная, и появляется в случайном месте поля, выровненном по

сетке шага (`random.randrange()` с шагом `RAZMER_SHAGA`, чтобы яблоко всегда оказывалось «в клетке»):

yabloko.py

```
yabloko = turtle.Turtle()
yabloko.speed(0)
yabloko.shape("circle")
yabloko.color("red")
yabloko.penup()

def novoe_yabloko():
    x = random.randrange(-GRANICA,
GRANICA, RAZMER_SHAGA)
    y = random.randrange(-GRANICA,
GRANICA, RAZMER_SHAGA)
    yabloko.goto(x, y)

novoe_yabloko()
```

Регистрирует ли экран нажатия клавиш со стрелками?

Подключаем клавиатуру и заставляем голову двигаться — с защитой от мгновенного разворота на 180°.

Регистрирует ли экран нажатия клавиш со стрелками?

Чтобы Turtle реагировал на клавиатуру, ему нужно явно разрешить «слушать» события (`screen.listen()`) и связать конкретные клавиши с функциями через `onkeypress()` — идея та же, что и `.bind()` у Tkinter в главе 17, только с более простым, специализированным интерфейсом:

klavishi.py

```
def idti_vverh():
    global napravlenie
    if napravlenie != "down":   # нельзя развернуться на 180° мгновенно
        napravlenie = "up"

def idti_vniz():
    global napravlenie
    if napravlenie != "up":
        napravlenie = "down"

# idti_vlevo() и idti_vpravo() устроены аналогично

screen.listen()
screen.onkeypress(idti_vverh, "Up")
screen.onkeypress(idti_vniz, "Down")
screen.onkeypress(idti_vlevo, "Left")
screen.onkeypress(idti_vpravo, "Right")
```

Почему нельзя просто развернуться на 180°?

Если змейка двигалась вправо, а игрок нажмёт «влево», голова мгновенно «наедет» на первый сегмент собственного тела — это выглядело бы как мгновенный проигрыш без видимой причины. Проверка `if napravlenie != "down":` запрещает разворот на месте, разрешая только повороты на 90°.

Заставляем голову змейки двигаться

dvizhenie_golovy.py

```
def dvigat_golovu():
    if napravlenie == "up":
        golova.sety(golova.ycor() +
RAZMER_SHAGA)
    elif napravlenie == "down":
        golova.sety(golova.ycor() -
RAZMER_SHAGA)
    elif napravlenie == "left":
        golova.setx(golova.xcor() -
RAZMER_SHAGA)
    elif napravlenie == "right":
        golova.setx(golova.xcor() +
RAZMER_SHAGA)
```

sety()/setx() — половина от goto()

`sety(y)` меняет только координату Y, оставляя X прежним — удобнее, чем вычислять обе координаты через `goto()` каждый раз.

Запускаем табло счёта

Отдельная черепашка без формы, которая только показывает текст текущего счёта.

Счёт удобно показывать отдельной черепашкой без формы (`hideturtle()`), которая ничего не рисует, кроме текста (`write()` из главы 7):

tablo_scheta.py

```
tablo = turtle.Turtle()
tablo.speed(0)
tablo.color("white")
tablo.penup()
tablo.hideturtle()
tablo.goto(0, 260)

def obnovit_tablo():
    tablo.clear()   # стираем старую
надпись перед новой
    tablo.write(f"Счёт: {schet}",
align="center", font=("Arial", 16,
"normal"))

obnovit_tablo()
```

clear() перед write() — обязательно

Без `tablo.clear()` каждый новый счёт накладывался бы поверх предыдущего — цифры быстро превратились бы в нечитаемую кашу.

Наша змейка ест!

Съеденное яблоко добавляет сегмент — и каждый сегмент должен правильно следовать за предыдущим.

Наша змейка ест!

Проверяем, достаточно ли близко голова оказалась к яблоку (метод `.distance()` считает расстояние между двумя

черепашками) — если да, яблоко «съедено»: появляется новое яблоко, змейка растёт, счёт увеличивается:

eda.py

```
def proverit_edu():
    global schet
    if golova.distance(yabloko) <
RAZMER_SHAGA:
        novoe_yabloko()
        dobavit_segment()
        schet += 10
        obnovit_tablo()
```

Заставляем двигаться всю змейку

Каждый сегмент тела должен занять место *предыдущего* сегмента (а первый — место головы) — это создаёт эффект «змейка ползёт целиком», а не «части двигаются независимо»:

dobavit_segment.py

```
def dobavit_segment():
    novyj = turtle.Turtle()
    novyj.speed(0)
    novyj.shape("square")
    novyj.color("grey")
    novyj.penup()
    segmenty.append(novyj)
```

dvigat_telo.py

```
def dvigat_telo():
    # начинаем с хвоста и идём к голове, чтобы не потерять позиции по пути
    for indeks in range(len(segmenty) - 1, 0, -1):
        x = segmenty[indeks - 1].xcor()
        y = segmenty[indeks - 1].ycor()
        segmenty[indeks].goto(x, y)

    if segmenty:
        segmenty[0].goto(golova.xcor(), golova.ycor())
```

Порядок вызовов решает всё

`dvigat_telo()` обязательно вызывается **до** `dvigat_golovu()` — иначе первый сегмент «унаследует» уже *новую* позицию головы вместо старой, и тело слипнется с головой в одну точку.

Проверка столкновений

Игра заканчивается при столкновении со стеной или с собственным хвостом.

Игра заканчивается при одном из двух столкновений: со стеной поля или с собственным телом.

stolknoveniya.py

```
def proverit_stolknoveniya():
    global igra_okonchena

    # столкновение со стеной
    if abs(golova.xcor()) > GRANICA or
abs(golova.ycor()) > GRANICA:
        igra_okonchena = True

    # столкновение с собственным телом
    for segment in segmenty:
        if segment.distance(golova) <
RAZMER_SHAGA / 2:
            igra_okonchena = True
```

abs() снова экономит код

`abs(golova.xcor()) > GRANICA` проверяет сразу обе стены (левую и правую) одним сравнением — вместо `golova.xcor() > GRANICA or golova.xcor() < -GRANICA`. Тот же приём мы использовали для факториала и других вычислений в главе 5.

★★ Самостоятельная задача

Ускорение игры

Увеличивайте скорость движения (уменьшайте задержку между шагами) на каждые 50 очков — игра станет постепенно сложнее.

Полный код

Собираем все части в единый игровой цикл — и подводим итоги главы.

Соберём один игровой шаг из всех частей главы — эта функция и есть сердце игры, вызываемое снова и снова, пока игра не закончится:

igrovoj_shag.py

```
def igrovoj_shag():
    if igra_okonchena:
        return False

    dvigat_telo()
    dvigat_golovu()
    proverit_edu()
    proverit_stolknoveniya()
    screen.update()
    return not igra_okonchena

def glavnyj_cikl():
    global napravlenie
    napravlenie = "right"
    while igrovoj_shag():
        screen.update()
    tablo.goto(0, 0)
    tablo.write(f"Игра окончена! Счёт:
{schet}", align="center", font=("Arial",
20, "bold"))
```

Полная, уже собранная и проверенная программа доступна отдельным файлом:

projects/turtle/snake/snake.py

Запустите игру у себя

`python snake.py` в терминале — управление стрелками клавиатуры.

★★★ Задача повышенной сложности

Уровни сложности

Добавьте выбор уровня сложности перед стартом игры (input() из главы 8) — влияющий на начальную скорость через задержку между шагами.

Итоги

Что мы узнали в этой главе

- `screen.tracer(0)` + ручной `screen.update()` — стандартный приём для плавной анимации в играх на Turtle.
- `screen.onkeypress()` связывает клавиши со стрелками с функциями изменения направления; запрет разворота на 180° предотвращает мгновенное самостолкновение.
- Тело змейки — список отдельных сегментов, каждый из которых следует за предыдущим; порядок

обновления (сначала тело, потом голова) критически важен.

- `.distance()` между двумя черепашками — удобный способ проверить, находятся ли они «достаточно близко» (для еды и столкновений).
- Игровой цикл — функция, вызываемая снова и снова, пока не наступит условие конца игры.

Станьте разработчиком игр с Pygame

Новый инструмент для игр — быстрее и гибче Turtle, с настоящим игровым циклом и обработкой кадров.

Что такое Pygame?

Специализированный инструмент для игр — с более тонким контролем над кадрами анимации, чем у Turtle и Tkinter.

Что такое Pygame?

`Pygame` — библиотека специально для игр: она даёт гораздо больше контроля над кадрами анимации, столкновениями и звуком, чем Turtle (главы 6–7, 12) или Tkinter (главы 16–19). В отличие от них, `Pygame` не входит в стандартную поставку Python — его нужно установить отдельно.

Устанавливаем и импортируем Pygame

ustanovka.txt

```
pip install pygame
```

Актуально на момент написания книги: используйте pygame-ce

На Python 3.14 у классического пакета `pygame` пока может не быть готового установочного файла (wheel) — установка попытается собрать его из исходного кода и упадёт с ошибкой про `sdl-config`. Решение — установить активно поддерживаемый форк с полностью совместимым API: `pip install pygame-ce`. Весь код в этой книге по-прежнему пишется как `import pygame` — оба пакета используют одно и то же имя модуля.

import_pygame.py

```
import pygame

pygame.init()   # обязательная
инициализация перед началом работы
```

Настраиваем игровой экран!

Игровой цикл — сердце любой игры на Pygame: он пишется вручную, кадр за кадром.

Настраиваем игровой экран!

Экран Pygame — обычная поверхность (`Surface`) заданного размера:

igrovoj_ekran.py

```
import pygame

pygame.init()
screen = pygame.display.set_mode((600,
400))
pygame.display.set_caption("Моя первая
игра")
```

Игровой цикл

В отличие от Tkinter с его `mainloop()`, в Pygame игровой цикл пишут вручную — это даёт полный контроль над тем, что происходит на каждом кадре:

igrovoj_cikl.py

```
clock = pygame.time.Clock()
rabotaet = True

while rabotaet:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            rabotaet = False

    pygame.display.flip()   # показать
нарисованный кадр
    clock.tick(60)          # не больше
60 кадров в секунду

pygame.quit()
```

Зачем нужен pygame.QUIT?

Без обработки события `pygame.QUIT` (нажатие на крестик окна) программа не узнает, что пользователь хочет закрыть игру, и окно перестанет отвечать. Обработка событий в начале каждого кадра цикла — обязательная часть любой программы на Pygame.

Делаем экран красивым

`screen.fill(цвет)` закрашивает весь экран цветом (в виде кортежа RGB, глава 11) — обычно первая команда в каждом кадре, иначе на экране останутся «следы» от предыдущего кадра:

zalivka_fona.py

```
CVET_FONA = (20, 20, 40)

screen.fill(CVET_FONA)
```

Создаём персонажей на экране

Первые персонажи — простые фигуры, рисуемые заново на каждом кадре.

Персонажей и объекты в Pygame рисуют прямо на экране каждый кадр — простыми фигурами (кругами, прямоугольниками), как мы делали в Turtle, или готовыми изображениями.

Начнём с фигур — они не требуют внешних файлов и отлично подходят для первых игр:

personazhi.py

```
CVET_IGROKA = (100, 200, 255)
CVET_VRAGA = (255, 80, 80)

x_igroka, y_igroka = 300, 350
x_vraga, y_vraga = 300, 50

# внутри игрового цикла, после
screen.fill(...):
pygame.draw.rect(screen, CVET_IGROKA,
(x_igroka, y_igroka, 40, 40))
pygame.draw.circle(screen, CVET_VRAGA,
(x_vraga, y_vraga), 20)
```

`pygame.draw.rect()` принимает кортеж `(x, y, ширина, высота)` — `(x, y)` это левый верхний угол, как у `Canvas` в главе 18, а не центр, как у `circle()`.

Rect — прямоугольник как объект

Для более сложных игр Pygame предлагает класс `pygame.Rect` — он хранит позицию и размер вместе и умеет проверять пересечения (`.colliderect()`) — то, что понадобится для космического шутера в главе 21.

Перемещаем персонажей

Движение — это просто изменение координат на каждом кадре; клавиатуру можно проверять двумя разными способами.

Перемещаем персонажей

«Движение» в Pygame — это просто изменение переменных позиции на каждом кадре перед тем, как персонаж будет нарисован заново:

peremeshenie.py

```
x_igroka += skorost_x
pygame.draw.rect(screen, CVET_IGROKA,
(x_igroka, y_igroka, 40, 40))
```

События нажатия клавиш

Есть два способа узнать о клавиатуре. Первый — через события, как в `pygame.event.get()` (удобно для разовых действий вроде прыжка):

sobytiya_klavish.py

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_SPACE:
            print("Прыжок!")
```

Второй способ — проверка текущего состояния клавиш на каждом кадре (удобнее для непрерывного движения, например, персонажа влево-вправо):

sostoyanie_klavish.py

```
klavishi = pygame.key.get_pressed()
if klavishi[pygame.K_LEFT]:
    x_igroka -= 5
if klavishi[pygame.K_RIGHT]:
    x_igroka += 5
```

Какой способ выбрать?

События (`event.type == pygame.KEYDOWN`) подходят для действий «один раз за нажатие» — прыжок, выстрел, пауза. `get_pressed()` подходит для непрерывного движения, пока клавиша зажата, — именно так реализовано управление кораблём в главе 21.

Мини-проект — прыгающий мяч

Первая настоящая мини-игра на Pygame — движение, отскоки и игровой цикл в одном коротком проекте.

Соберём всё в один мини-проект: мяч, который отскакивает от всех четырёх стен экрана.

prygayushij_myach.py

```
import pygame

SHIRINA, VYSOTA = 600, 400
RADIUS = 20

pygame.init()
screen =
pygame.display.set_mode((SHIRINA,
VYSOTA))
clock = pygame.time.Clock()

x, y = SHIRINA // 2, VYSOTA // 2
dx, dy = 4, 3

rabotaet = True
while rabotaet:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            rabotaet = False

    x += dx
    y += dy

    if x - RADIUS < 0 or x + RADIUS >
SHIRINA:
        dx = -dx
```

```
    if y - RADIUS < 0 or y + RADIUS >
VYSOTA:
        dy = -dy

    screen.fill((20, 20, 40))
    pygame.draw.circle(screen, (255,
100, 100), (int(x), int(y)), RADIUS)
    pygame.display.flip()
    clock.tick(60)

pygame.quit()
```

dx = -dx — отражение одной строкой

Смена знака скорости на противоположный — стандартный приём отражения от стены: движение продолжается с той же скоростью, только в обратную сторону, без дополнительных проверок направления.

Полный, уже проверенный файл — отдельно:

projects/pygame/bouncing-ball/bouncing_ball.py

★★ Самостоятельная задача

Меняем цвет при отскоке

При каждом отскоке от стены выбирайте новый случайный цвет мяча (random.choice() из главы 5/11).

★★★ Задача повышенной сложности

Два мяча

Добавьте второй мяч с собственными x, y, dx, dy — оба должны двигаться и отскакивать независимо.

Итоги

Что мы узнали в этой главе

- Pygame не входит в стандартную поставку Python — устанавливается через `pip install`.
- Игровой цикл в Pygame пишут вручную: обработка событий → обновление позиций → перерисовка → `clock.tick(FPS)`.
- `screen.fill()` в начале каждого кадра предотвращает «следы» от предыдущих кадров.
- Персонажей рисуют заново на каждом кадре — движение реализуется просто

изменением координат перед отрисовкой.

- Клавиатуру проверяют либо через события (разовые действия), либо через `pygame.key.get_pressed()` (непрерывное движение).

ГЛАВА 21 · СТР. 467

Проект: космический шутер с Pygame

Самая крупная игра книги — корабль, враги, стрельба, счёт и конец игры, собранные шаг за шагом.

Игра «Космический шутер»

Самая крупная игра книги — все части уже знакомы, теперь они работают вместе.

Игра «Космический шутер»

Самый крупный проект книги: корабль игрока внизу экрана, враги, спускающиеся сверху, возможность стрелять и уничтожать врагов, счёт очков и полноценный конец игры. Каждая часть уже знакома по предыдущим главам — просто теперь они работают вместе.

Импортируем необходимые модули

importy.py

```
import random
import pygame
```

Инициализируем всё необходимое

inicializaciya.py

```
SHIRINA, VYSOTA = 500, 600
FPS = 60

pygame.init()
screen =
pygame.display.set_mode((SHIRINA,
VYSOTA))
pygame.display.set_caption("Космический
шутер")
clock = pygame.time.Clock()
shrift = pygame.font.SysFont(None, 32)
```

pygame.font — ещё один модуль Pygame

`pygame.font.SysFont(None, 32)` создаёт шрифт системным по умолчанию, размером 32 — понадобится для табло счёта в разделе 21.5.

Игровой цикл

План того, что должно происходить на каждом кадре — и первый персонаж игры.

Игровой цикл

Как и в главе 20, цикл — сердце игры. На этот раз он будет обновлять сразу несколько вещей на каждом кадре: положение корабля, врагов, пуль — и проверять столкновения:

igrovoj_cikl_plan.py

```
while rabotaet:
    # 1. обработать события (выход,
выстрел)
    # 2. обработать зажатые клавиши
(движение корабля)
    # 3. обновить положение врагов и
пуль, проверить столкновения
    # 4. нарисовать всё заново
    # 5. clock.tick(FPS)
```

Создаём космический корабль

Вместо отдельных переменных x, y корабля удобно использовать `pygame.Rect` — он сразу хранит позицию и размер вместе и понадобится для проверки столкновений:

korabl.py

```
KORABL_SHIRINA, KORABL_VYSOTA = 50, 40

korabl = pygame.Rect(
    SHIRINA // 2 - KORABL_SHIRINA // 2,
    VYSOTA - KORABL_VYSOTA - 20,
    KORABL_SHIRINA,
    KORABL_VYSOTA,
)

# в игровом цикле:
pygame.draw.rect(screen, (80, 220, 120),
korabl)
```

Rect хранит четыре числа — и много полезных свойств

`Rect(x, y, ширина, высота)` дополнительно вычисляет `.centerx`, `.bottom`, `.top` и другие удобные координаты — они пригодятся уже в следующем разделе.

Перемещаем космический корабль

Управление с ограничением по краям экрана — и первые враги, появляющиеся сверху.

Перемещаем космический корабль

Управление — через `get_pressed()` из главы 20 (непрерывное движение, пока клавиша зажата), с ограничением, чтобы корабль не улетел за края экрана:

dvizhenie_korablya.py

```
KORABL_SKOROST = 6

def obrabotat_klavishi(korabl, klavishi):
    if klavishi[pygame.K_LEFT]:
        korabl.x -= KORABL_SKOROST
    if klavishi[pygame.K_RIGHT]:
        korabl.x += KORABL_SKOROST
    korabl.x = max(0, min(korabl.x,
SHIRINA - KORABL_SHIRINA))
```

max/min — тот же приём ограничения, что и в главе 20

`max(0, min(korabl.x, SHIRINA - KORABL_SHIRINA))` гарантирует, что `korabl.x` никогда не выйдет за пределы `[0, SHIRINA - KORABL_SHIRINA]` — тот же самый приём «зажимания» значения в диапазон, что мы использовали для прыгающего мяча.

Создаём и перемещаем врагов

Врагов будет много, и они появляются со временем — значит, нужен список (глава 11) и счётчик кадров до следующего появления:

vragi.py

```
VRAG_SHIRINA, VRAG_VYSOTA = 40, 30
VRAG_SKOROST = 2
INTERVAL_POYAVLENIYA_VRAGA = 45   #
кадров между новыми врагами

vragi = []
kadrov_do_vraga =
INTERVAL_POYAVLENIYA_VRAGA

def sozdat_vraga():
    x = random.randint(0, SHIRINA -
VRAG_SHIRINA)
    return pygame.Rect(x, -VRAG_VYSOTA,
VRAG_SHIRINA, VRAG_VYSOTA)

# в игровом цикле, каждый кадр:
kadrov_do_vraga -= 1
if kadrov_do_vraga <= 0:
    vragi.append(sozdat_vraga())
    kadrov_do_vraga =
INTERVAL_POYAVLENIYA_VRAGA

for vrag in vragi:
    vrag.y += VRAG_SKOROST
```

y = -VRAG_VYSOTA — враг появляется чуть выше экрана

Отрицательная стартовая координата Y означает, что враг рождается чуть выше видимой области и плавно «въезжает» в кадр сверху — а не появляется резко посередине экрана.

Стреляем

Пробел создаёт пулю у носа корабля — она улетает вверх, пока не покинет экран.

Пули — тоже список `Rect`, появляющийся у носа корабля при нажатии пробела и улетающий вверх на каждом кадре:

strelba.py

```
PULYA_SHIRINA, PULYA_VYSOTA = 4, 12
PULYA_SKOROST = 9

puli = []

def vystrelit():
    pulya = pygame.Rect(
        korabl.centerx -
PULYA_SHIRINA // 2,
        korabl.top,
        PULYA_SHIRINA,
        PULYA_VYSOTA,
    )
    puli.append(pulya)

# в обработке событий:
if event.type == pygame.KEYDOWN and
event.key == pygame.K_SPACE:
    vystrelit()

# в обновлении кадра:
for pulya in puli:
    pulya.y -= PULYA_SKOROST
```

```
puli = [p for p in puli if p.bottom >
0]   # убираем улетевшие за экран
```

Список через генератор списков — чистка «мусора»

`[p for p in puli if p.bottom > 0]` (генератор списков из главы 11) оставляет только те пули, что ещё видны на экране — без этой строки список пуль рос бы бесконечно, замедляя игру всё сильнее.

Запускаем табло счёта

Текст в Pygame рисуется в два шага: сначала render(), потом blit() на экран.

Счёт выводится готовым шрифтом Pygame (`pygame.font`, раздел 21.1) — в отличие от Turtle, здесь текст сначала «отрисовывается» в отдельное изображение (`render()`), а затем накладывается на экран (`blit()`):

tablo_scheta.py

```
schet = 0

# в отрисовке кадра:
tablo = shrift.render(f"Счёт: {schet}",
True, (255, 255, 255))
screen.blit(tablo, (10, 10))
```

render() + blit() — тот же принцип, что write() у Turtle

`render(текст, сглаживание, цвет)` создаёт изображение текста; `screen.blit(изображение, позиция)` накладывает его на экран — концептуально то же самое, что `artist.write()` у Turtle из главы 7, просто в два явных шага вместо одного.

Уничтожаем врагов

colliderect() проверяет столкновения — попадание уничтожает врага, а враг может уничтожить корабль.

Уничтожаем врагов

Столкновение пули с врагом проверяет готовый метод `Rect.colliderect()` — при попадании оба удаляются из своих списков, а счёт увеличивается:

unichtozhenie_vragov.py

```
novye_puli = []
novye_vragi = list(vragi)
for pulya in puli:
    popala = False
    for vrag in list(novye_vragi):
        if pulya.colliderect(vrag):
            novye_vragi.remove(vrag)
            schet += 10
            popala = True
            break
    if not popala:
        novye_puli.append(pulya)
puli = novye_puli
vragi = novye_vragi
```

Почему не удалять элементы прямо во время перебора списка?

Удаление элемента из списка `vragi` прямо внутри цикла `for vrag in vragi:` может пропустить соседние элементы — список «сдвигается» под ногами цикла. Безопаснее собрать **новые** списки уцелевших пуль и врагов, как показано выше.

Уничтожаем космический корабль!

Если враг долетел до низа экрана или столкнулся с кораблём — игра заканчивается:

unichtozhenie_korablya.py

```
for vrag in vragi:
    if vrag.bottom >= VYSOTA or
vrag.colliderect(korabl):
        igra_okonchena = True
        break
```

Перерисовываем врагов

Каждый кадр рисуется заново целиком — а при окончании игры на экране появляется финальная надпись.

Перерисовываем врагов

После всех проверок и обновлений кадр рисуется заново целиком — фон, корабль, пули, враги и счёт, в таком порядке (чтобы более поздние элементы оказывались поверх более ранних):

pererisovka.py

```
screen.fill((10, 10, 20))
pygame.draw.rect(screen, (80, 220, 120),
korabl)
for pulya in puli:
    pygame.draw.rect(screen, (240, 220,
80), pulya)
for vrag in vragi:
    pygame.draw.rect(screen, (230, 60,
60), vrag)

tablo = shrift.render(f"Счёт: {schet}",
True, (255, 255, 255))
screen.blit(tablo, (10, 10))

pygame.display.flip()
```

Игра окончена!

Когда `igra_okonchena` становится истинной, вместо обычной игровой логики выводится финальный экран:

game_over.py

```
if igra_okonchena:
    nadpis = shrift_bolshoj.render("ИГРА
ОКОНЧЕНА", True, (255, 255, 255))
    rect =
nadpis.get_rect(center=(SHIRINA // 2,
VYSOTA // 2))
    screen.blit(nadpis, rect)
```

get_rect(center=...) — удобное центрирование текста

Вместо ручного вычисления координат текста по его ширине и высоте, `get_rect(center=(x, y))` сам находит нужный левый верхний угол так, чтобы текст оказался центрирован ровно в точке `(x, y)`.

Полный код

Вся игра целиком — уже собранная и проверенная — и подведение итогов главы.

Полная, уже собранная и проверенная игра — отдельным файлом:

projects/pygame/space-shooter/space_shooter.py

Запустите игру у себя

`python space_shooter.py` в терминале. Управление: стрелки влево/вправо — движение, пробел — выстрел.

★★ Самостоятельная задача

Жизни вместо мгновенного конца

Добавьте переменную zhizni = 3 — при столкновении корабля с врагом отнимайте одну жизнь и убирайте врага, вместо немедленного окончания игры.

★★★ Задача повышенной сложности

Уровни сложности

Постепенно уменьшайте INTERVAL_POYAVLENIYA_VRAGA по мере роста счёта — враги должны появляться всё чаще.

Итоги

Что мы узнали в этой главе

- Крупный проект строится из уже знакомых, более мелких приёмов — корабль, враги и пули устроены похоже, просто с разными правилами движения.
- `pygame.Rect` хранит позицию и размер вместе и умеет проверять столкновения через `.colliderect()`.
- При удалении элементов по условию безопаснее собрать новый список уцелевших элементов, чем изменять список во время перебора.

- Текст в Pygame рисуется в два шага: `render()` создаёт изображение, `blit()` накладывает его на экран.
- Порядок отрисовки элементов кадра определяет, что окажется «поверх» — рисуйте фон первым, интерфейс (счёт, финальный экран) — последним.

ГЛАВА 22 · СТР. 497

Веб-разработка с Python

Как устроен веб, из чего состоит сайт и как Python с помощью Flask превращается в сервер.

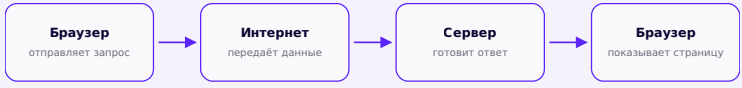
Python и веб-разработка

Каждый сайт — это диалог между браузером и сервером. Разберёмся, как он устроен.

Python и веб-разработка

Каждый раз, когда вы открываете сайт в браузере, происходит короткий диалог между двумя программами: браузером (**клиентом**) и другой программой,

которая отвечает на его запросы (**сервером**). Браузер спрашивает — сервер отвечает:



Путь запроса от браузера до сервера и обратно

Этот диалог называется **HTTP** (HyperText Transfer Protocol — протокол передачи гипертекста). Браузер отправляет HTTP-запрос («дай мне главную страницу»), сервер отправляет HTTP-ответ (обычно — HTML-страницу).

Фронтенд и бэкенд

То, что видит и с чем взаимодействует пользователь в браузере — HTML, CSS, JavaScript — называют **фронтендом** (front — «перед»). Программа, которая работает на сервере и готовит ответы — **бэкенд** (back — «зад», «тыл»). Python на сайтах почти всегда работает именно как бэкенд.

В этой главе мы кратко познакомимся со всеми тремя языками фронтенда — HTML, CSS и JavaScript, — а затем напишем свой первый бэкенд на Python с помощью библиотеки **Flask**.

Строительные блоки — HTML

HTML описывает структуру страницы: где заголовок, где текст, где ссылка.

Строительные блоки — HTML

HTML (HyperText Markup Language — язык гипертекстовой разметки) описывает *структуру* страницы: где заголовок, где текст, где картинка, где ссылка. Каждый кусочек содержимого оборачивается в **тег**:

stranica.html

```
<!doctype html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <title>Моя первая страница</title>
</head>
<body>
  <h1>Привет, мир!</h1>
  <p>Это мой первый сайт на HTML.</p>
  <ul>
    <li>Учу Python</li>
    <li>Учу HTML</li>
  </ul>
  <a href="https://
python.org">Официальный сайт Python</a>
</body>
</html>
```

Теги обычно идут парами

`<h1>` открывает заголовок, `</h1>` (со слэшем) его закрывает — почти как открывающая и закрывающая скобки в Python. Всё, что между ними, и есть содержимое этого элемента.

Самые частые теги: `<h1>` … `<h6>` — заголовки разного уровня, `<p>` — абзац текста, `<ul>` / `<li>` — список, `<a href="…">` — ссылка, `<img src="…">` — картинка.

Делаем красивее — CSS

CSS превращает голый HTML в страницу с цветом, шрифтами и аккуратными отступами.

Делаем красивее — CSS

Сам по себе HTML выглядит скучно — чёрный текст на белом фоне. За внешний вид отвечает **CSS** (Cascading Style Sheets — каскадные таблицы стилей). CSS выбирает элементы и задаёт им свойства — цвет, отступы, размер шрифта:

stili.css

```
body {
  font-family: sans-serif;
  background: #f7f7fb;
  color: #1a1a2e;
}

h1 {
  color: #4a2fbd;
}

p {
  line-height: 1.6;
}
```

Подключить CSS-файл к HTML-странице можно одной строкой внутри `<head>`:

podklyuchenie.html

```
<link rel="stylesheet" href="stili.css">
```

«Каскадные» — откуда название

Если два правила CSS задают один и тот же элемент по-разному, побеждает то, что идёт *позже* или более точно указывает на элемент — стили как бы «стекают» друг на друга, как каскад. Именно поэтому в имени CSS есть слово «каскадные».

Динамический фронтенд — JavaScript

Если HTML — это скелет, а CSS — одежда, то JavaScript — это движения страницы.

Динамический фронтенд — JavaScript

HTML описывает структуру, CSS — оформление, а **JavaScript** добавляет *поведение*: реакцию на клики, изменение текста без перезагрузки страницы, проверку форм. JavaScript выполняется прямо в браузере пользователя:

povedenie.html

```
<button id="knopka">Нажми меня</button>
<p id="tekst">Пока ничего не произошло.</
p>

<script>
  const knopka =
document.getElementById("knopka");
  const tekst =
document.getElementById("tekst");

  knopka.addEventListener("click",
function () {
    tekst.textContent = "Кнопку нажали!";
  });
</script>
```

JavaScript и Python — похожи, но разные

Синтаксис отличается (фигурные скобки вместо отступов, `const` / `function` вместо привычных конструкций Python), но идеи те же: переменные, функции, условия, циклы. Если вы понимаете Python — освоить основы JavaScript будет намного проще.

Три языка вместе — HTML, CSS и JavaScript — и есть тот самый «фронтенд», который видит и с которым взаимодействует пользователь. Теперь перейдём к тому, что происходит на сервере — то есть к Python.

Flask в Python

Flask превращает программу на Python в веб-сервер — и делает это на удивление просто.

Flask в Python

Flask — это лёгкая библиотека, которая превращает программу на Python в веб-сервер: она принимает HTTP-запросы от браузера и отправляет в ответ HTML-страницы. Установка — как у любой библиотеки:

terminal.txt

```
pip install flask
```

Самое маленькое Flask-приложение

app_minimal.py

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def glavnaya():
    return "Привет, мир!"

if __name__ == "__main__":
    app.run(debug=True)
```

@app.route("/") — декоратор маршрута

Строчка над функцией — **декоратор** (глава 15 показывала похожую идею). `@app.route("/")` говорит Flask: «когда браузер запросит адрес `/`, вызови функцию `glavnaya()` и отправь то, что она вернёт».

Шаблоны — HTML с вставками Python

Возвращать HTML прямо строкой в Python неудобно. Flask использует движок шаблонов **Jinja2** — обычный HTML-файл, куда можно вставлять python-подобные выражения в фигурных скобках `{{ }}`:

templates/index.html

```
<h1>Мой список задач</h1>
<ul>
  {% for zadacha in zadachi %}
    <li>{{ zadacha }}</li>
  {% endfor %}
</ul>
```

app.py

```
from flask import Flask, render_template

app = Flask(__name__)
zadachi = ["Выучить основы Python",
"Собрать сайт на Flask"]

@app.route("/")
def glavnaya():
    return render_template("index.html",
zadachi=zadachi)
```

Динамические адреса

Часть адреса можно превратить в параметр функции — Flask сам передаст в неё нужное значение из URL:

dinamicheskij_marshrut.py

```
@app.route("/privet/<imya>")
def privet(imya):
    return
render_template("privet.html", imya=imya)
```

/privet/Сергей → imya = "Сергей"

Открыв в браузере адрес `/privet/Сергей`, вы получите в переменной `imya` строку `"Сергей"` — Flask сам достаёт эту часть из адреса.

Принимаем данные из формы

Чтобы пользователь мог что-то отправить на сервер (например, добавить задачу), используется HTML-форма и маршрут, принимающий метод `POST`:

dobavlenie.py

```
from flask import request, redirect,
url_for

@app.route("/dobavit", methods=["POST"])
def dobavit():
    novaya_zadacha =
request.form.get("zadacha", "").strip()
    if novaya_zadacha:
        zadachi.append(novaya_zadacha)
    return redirect(url_for("glavnaya"))
```

Полный проект

Всё вместе — маршруты, шаблоны, форма и статический CSS-файл — собрано в готовый мини-сайт «Список задач»:

projects/flask/todo-app/app.py

Запустите сайт у себя

`python app.py` в терминале внутри `projects/flask/todo-app/`, затем откройте `http://127.0.0.1:5000/` в браузере.

★★ Самостоятельная задача

Счётчик задач

Добавьте в index.html строку, показывающую общее количество задач: {{ zadachi|length }}.

★★★ Задача повышенной сложности

Удаление задачи

Добавьте маршрут /udalit/<int:indeks>, который удаляет задачу по её номеру в списке и делает редирект на главную.

Итоги

От HTTP-запроса до собственного сайта на Flask — коротко о том, что мы прошли.

Что мы узнали в этой главе

- Сайт — это диалог по протоколу HTTP: браузер (клиент) отправляет запрос, сервер отвечает.
- Фронтенд (то, что видит пользователь) строится из трёх языков: HTML (структура), CSS (оформление) и JavaScript (поведение).
- Бэкенд — программа на сервере, которая готовит ответы. На Python для этого часто используют библиотеку Flask.

- `@app.route("/адрес")` связывает адрес с функцией, которая на него отвечает.
- Шаблоны Jinja2 позволяют вставлять данные Python прямо в HTML через `{{ }}` и `{% %}`.
- Формы с методом POST и `request.form` позволяют принимать данные, которые пользователь ввёл в браузере.

Веб-разработка — огромная тема, и эта глава лишь приоткрыла дверь. Если вам понравилось — в главе 24 («Что дальше?») есть раздел с идеями, куда двигаться дальше.

ГЛАВА 23 · СТР. 511

Ещё больше мини-проектов

Шесть небольших, но полноценных проектов — от калькулятора до собственного файла с заметками.

Проект 23-1: Калькулятор с Tkinter

Экран, сетка кнопок и функция vychislit_vyrazhenie() — вот и весь калькулятор.

Проект 23-1: Калькулятор с Tkinter

Собираем настоящий калькулятор: экран сверху, кнопки с цифрами и знаками снизу. Идея та же, что и в тренажёре «Крестики-нолики» из главы 19 — сетка кнопок, каждая из которых вызывает одну и ту же функцию с разным аргументом:

sozdanie_knopok.py

```
KNOPKI = [
    ("7", 1, 0), ("8", 1, 1), ("9", 1,
2), ("/", 1, 3),
    ("4", 2, 0), ("5", 2, 1), ("6", 2,
2), ("*", 2, 3),
    # ...
]

for podpis, stroka, stolbec in KNOPKI:
    knopka = tk.Button(root, text=podpis,
                       command=lambda
s=podpis: na_cifru_ili_znak_nazhali(s))
    knopka.grid(row=stroka,
column=stolbec, sticky="we")
```

lambda s=podpis — почему нельзя просто lambda: ...(podpis)

Это тот же самый «капкан позднего связывания замыканий», что мы разбирали в главе 17: без `s=podpis` все кнопки запомнили бы одну и ту же, последнюю по циклу переменную `podpis` — и нажатие любой кнопки вставляло бы один и тот же символ.

Вычисляем выражение

Вместо того чтобы писать свой разбор арифметики, используем встроенную функцию `eval()` — но **осторожно**:

проверяем, что в строке нет ничего, кроме цифр и знаков, и запрещаем доступ к встроенным функциям Python:

vychislenie.py

```
def vychislit_vyrazhenie(vyrazhenie):
    dopustimye_simvoly =
set("0123456789+-*/(). ")
    if not vyrazhenie or not
set(vyrazhenie) <= dopustimye_simvoly:
        return "Ошибка"
    try:
        return str(eval(vyrazhenie,
{"__builtins__": {}}, {}))
    except (SyntaxError,
ZeroDivisionError, ValueError):
        return "Ошибка"
```

Почему просто не написать eval(vyrazhenie)?

Обычный `eval()` выполнит *любой* Python-код, который ему передать — например, команду для удаления файлов. Проверка разрешённых символов и пустой словарь `{"__builtins__": {}}` не дают выражению сделать ничего, кроме простой арифметики.

Полный код: projects/tkinter/calculator/calculator.py

Проект 23-2: Генератор случайных историй

random.choice() пять раз подряд — и шаблон превращается в маленькую историю.

Проект 23-2: Генератор случайных историй

Заполняем шаблон предложения случайно выбранными словами из нескольких списков — иногда получаются смешные истории

(«безумные библиотеки», mad libs). Вся идея — в `random.choice()` (глава 5) и методе строк `.format()` (глава 8):

story_generator.py

```
PRILAGATELNYE = ["храбрый",
"любопытный", "рассеянный", "весёлый",
"загадочный"]
SUSHESTVITELNYE = ["дракон",
"программист", "кот", "путешественник",
"робот"]
MESTA = ["в тёмном лесу", "на далёкой
планете", "в старой библиотеке"]
GLAGOLY = ["нашёл", "потерял",
"починил", "изобрёл", "испугался"]
PREDMETY = ["волшебный ноутбук",
"древний свиток", "сломанный компас"]

SHABLON = (
    "Однажды {prilagatelnoe}
{sushestvitelnoe} {mesto} {glagol}
{predmet}. "
    "С тех пор жизнь его больше не была
прежней."
)

def sluchajnaya_istoriya():
    return SHABLON.format(

prilagatelnoe=random.choice(PRILAGATELNYE),
```

```
sushestvitelnoe=random.choice(SUSHESTVITELNY
        mesto=random.choice(MESTA),
        glagol=random.choice(GLAGOLY),
        predmet=random.choice(PREDMETY),
    )
```

Пять независимых случайных выборов — 5×5×5×5×5 = 3125 историй

Даже с пятью короткими списками по 5 слов получается больше трёх тысяч различных комбинаций — вот почему такой простой приём кажется таким разнообразным.

Полный код: projects/console/story-generator/story_generator.py

★★ Самостоятельная задача

Ещё один список

Добавьте список НАРЕЧИЯ (например, "внезапно", "случайно", "незаметно") и вставьте {narechie} в шаблон.

Проект 23-3: Игра «Камень, ножницы, бумага»

Вся логика игры — один словарь: кто кого побеждает.

Проект 23-3: Игра «Камень, ножницы, бумага»

Классика! Вся логика игры умещается в одном словаре — кто кого побеждает:

logika_pobeditelya.py

```
POBEZHDAET = {
    "камень": "ножницы",
    "ножницы": "бумага",
    "бумага": "камень",
}

def opredelit_pobeditelya(hod_igroka,
hod_kompyutera):
    if hod_igroka == hod_kompyutera:
        return "ничья"
    if POBEZHDAET[hod_igroka] ==
hod_kompyutera:
        return "игрок"
    return "компьютер"
```

Словарь вместо длинной цепочки if/elif

Можно было написать девять условий `if hod_igroka == "камень" and hod_kompyutera == "ножницы": ...` — но словарь `POBEZHDAET` делает то же самое в трёх строчках и легко читается: «камень побеждает ножницы».

Ход компьютера и полный раунд

raund.py

```
def hod_kompyutera():
    return random.choice(VARIANTY)

def sygrat_raund(hod_igroka):
    hod_pk = hod_kompyutera()
    pobeditel =
opredelit_pobeditelya(hod_igroka, hod_pk)
    return hod_pk, pobeditel
```

Полный код (с меню и счётом):

projects/console/rock-paper-scissors/rps.py

★★★ Задача повышенной сложности

Добавляем ящерицу и Спока

Реализуйте расширенную версию игры «Камень, ножницы, бумага, ящерица, Спок» — понадобится словарь POBEZHDAET с пятью ключами, каждый побеждает по два варианта.

Проект 23-4: Отскакивающий от четырёх стен мяч

Та же физика, что и в главе 20, — но теперь мяч это класс, а значит, их может быть сколько угодно.

Проект 23-4: Отскакивающий от четырёх стен мяч с Pygame

В главе 20 мяч был устроен через обычные переменные и функции. Теперь, когда мы знаем классы (глава 15), опишем мяч как объект — и сразу получим возможность легко завести *несколько* мячей одновременно:

myach_klass.py

```
class Myach:
    def __init__(self, x, y, dx, dy,
radius, cvet):
        self.x = x
        self.y = y
        self.dx = dx
        self.dy = dy
        self.radius = radius
        self.cvet = cvet
        self.otskokov = 0

    def shag(self):
        self.x += self.dx
        self.y += self.dy
        if self.x - self.radius < 0 or
self.x + self.radius > SHIRINA:
            self.dx = -self.dx
            self.otskokov += 1
        # ... то же самое для self.y и
VYSOTA

    def narisovat(self):
        pygame.draw.circle(screen,
```

```
self.cvet, (int(self.x), int(self.y)),
self.radius)
```

Список объектов вместо списка переменных

Раньше пришлось бы заводить отдельные `x1, y1, dx1, dy1, x2, y2, dx2, dy2, …` для каждого мяча. Теперь достаточно `myachi = [Myach(...), Myach(...), Myach(...)]` — и цикл `for myach in myachi: myach.shag()` двигает их все разом.

Полный код: projects/pygame/bouncing-balls-oop/bouncing_balls.py

★★ Самостоятельная задача

Столкновение мячей друг с другом

Добавьте проверку расстояния между каждой парой мячей — если они соприкоснулись, поменяйте местами их dx и dy.

Проект 23-5: Преобразование температуры

Одна функция перевода в Цельсий — и все шесть направлений пересчёта готовы.

Проект 23-5: Приложение для преобразования температуры

Небольшое, но по-настоящему полезное приложение: вводим температуру, выбираем единицу через переключатели (`Radiobutton`) и сразу видим значение во всех трёх шкалах:

preobrazovanie.py

```
def celsij_v_farengejt(c):
    return c * 9 / 5 + 32

def celsij_v_kelvin(c):
    return c + 273.15

def preobrazovat(znachenie, iz_edinicy):
    if iz_edinicy == "C":
        c = znachenie
    elif iz_edinicy == "F":
        c = farengejt_v_celsij(znachenie)
    else:
        c = kelvin_v_celsij(znachenie)

    return {
        "C": c,
        "F": celsij_v_farengejt(c),
        "K": celsij_v_kelvin(c),
    }
```

Всегда переводим через Цельсий

Вместо шести отдельных формул (C→F, C→K, F→C, F→K, K→C, K→F) достаточно уметь переводить *в* Цельсий и *из* Цельсия — остальные пять переводов получаются сами собой через промежуточный шаг.

Переключатели Radiobutton

radiobutton.py

```
edinica = tk.StringVar(value="C")

for kod, podpis in [("C", "Цельсий"),
("F", "Фаренгейт"), ("K", "Кельвин")]:
    tk.Radiobutton(root, text=podpis,
variable=edinica, value=kod).grid(...)
```

Radiobutton — «выбери один из нескольких»

Все переключатели с одинаковой `variable=edinica` работают как группа — можно выбрать только один одновременно, а `edinica.get()` всегда возвращает значение выбранного.

Полный код: projects/tkinter/temperature-converter/temperature_converter.py

Проект 23-6: Знакомство с файлами и Tkinter

Простое приложение «Заметки» — и подведение итогов главы.

Проект 23-6: Знакомство с файлами и Tkinter

Последний мини-проект главы объединяет файлы (глава 14) и Tkinter (главы 16-19): простое приложение «Заметки» с текстовым полем и тремя кнопками — сохранить, загрузить, очистить:

sohranenie_zagruzka.py

```
FAJL_ZAMETOK = Path(__file__).parent /
"zametka.txt"

def sohranit_zametku():
    tekst = polye_teksta.get("1.0",
"end-1c")
    FAJL_ZAMETOK.write_text(tekst,
encoding="utf-8")
    status_text.set(f"Сохранено в
{FAJL_ZAMETOK.name}")

def zagruzit_zametku():
    if not FAJL_ZAMETOK.exists():

status_text.set("Файл заметки ещё не
создан — сначала сохраните.")
        return
    tekst =
FAJL_ZAMETOK.read_text(encoding="utf-8")
    polye_teksta.delete("1.0", "end")
    polye_teksta.insert("1.0", tekst)
```

"1.0" и "end-1c" — адресация текста в Text

У виджета `Text` (в отличие от `Entry`) текст адресуется парой «строка.столбец»: `"1.0"` — самое начало (строка 1, столбец 0), `"end-1c"` — конец текста без завершающего невидимого перевода строки, который Tkinter добавляет сам.

Проверка exists() перед чтением

Если попытаться прочитать несуществующий файл через `read_text()`, программа упадёт с `FileNotFoundError`. Проверка `if not FAJL_ZAMETOK.exists():` превращает падение программы в понятное сообщение для пользователя.

Полный код: projects/tkinter/notes-app/notes_app.py

★★ Самостоятельная задача

Предупреждение о несохранённых изменениях

Заведите переменную byli_izmeneniya = False, ставьте её в True при любом изменении текста и показывайте предупреждение при попытке «Очистить поле», если она True.

Итоги

Что мы узнали в этой главе

- Шесть небольших проектов показывают, как уже знакомые инструменты — Tkinter, Pygame, файлы, классы, словари — снова и снова складываются в новые приложения.
- Безопасный `eval()` с проверкой символов и пустым `__builtins__` позволяет вычислять арифметические выражения, не рискуя выполнить произвольный код.
- Словарь — отличный способ описать отношения между вариантами (кто кого

побеждает, что во что переводится), избегая длинных цепочек if/elif.

- Класс превращает набор связанных переменных (позиция, скорость, цвет мяча) в один объект — и сразу же позволяет завести их сколько угодно через список.
- Перед чтением файла стоит проверять `Path.exists()` — это превращает возможный крах программы в понятное сообщение для пользователя.

Что дальше?

Книга заканчивается — но путешествие в программирование только начинается. Куда двигаться дальше.

Идеи мини-проектов, которые можно попробовать

Все инструменты для этих идей вы уже освоили — осталось только начать.

Идеи мини-проектов, которые можно попробовать

Лучший способ закрепить всё, что вы прошли в этой книге, — придумывать и собирать собственные небольшие программы. Вот несколько идей, чтобы начать. Все инструменты для них у вас уже есть.

Приложение для конвертации валют

Похоже на конвертер температуры из главы 23, только вместо °C/°F/K — рубли, доллары и евро. Курсы валют можно на первых порах просто зашить в словарь прямо в коде (`KURSY = {"USD": 95.0, "EUR": 103.0}`), а позже, если захочется, — научиться получать их из интернета.

Гонка в Pygame

Машинка (прямоугольник или картинка) едет по дороге, стрелки влево/вправо двигают её между полосами, а сверху появляются препятствия — точно так

же, как враги в «Космическом шутере» из главы 21, только вместо стрельбы нужно уворачиваться.

Другие узоры в Turtle

Глава 12 показала спирали и мандалы — а что, если менять не только угол поворота, но и цвет пера на каждом шаге? Или рисовать не круги, а звёзды разного размера по кругу? Попробуйте взять код мандалы и заменить одну строчку — и посмотреть, что получится.

Идеи итоговых проектов, которые можно попробовать

Более крупные проекты, объединяющие приёмы сразу из нескольких глав книги.

Идеи итоговых проектов, которые можно попробовать

Если мини-проектов уже мало — вот идеи для проектов покрупнее, каждый из которых объединяет несколько приёмов из разных глав книги.

Игра «Змейка» в Pygame

В главе 19 змейка была на Turtle. Попробуйте собрать её заново на Pygame — движение списком сегментов (глава 11), столкновения через `colliderect()` (глава 21), еда, которая появляется в случайном месте (`random.randint()`, глава 5). Получится интересно сравнить два подхода к одной и той же игре.

Уклонись от пули

Обратная версия «Космического шутера» из главы 21: вместо того чтобы стрелять по врагам, нужно уворачиваться от падающих сверху

предметов, управляя персонажем внизу экрана. Чем дольше продержался — тем выше счёт.

Игра на память в Pygame

Классическая игра «Найди пару»: на экране — сетка перевёрнутых карточек, клик переворачивает две из них, если символы совпали — карточки остаются открытыми, если нет — переворачиваются обратно. Пригодятся двумерные списки (глава 11) и обработка кликов мыши (глава 20).

С чего начать любой из этих проектов

Не пытайтесь написать всё сразу. Разбейте проект на маленькие шаги — как делали все проекты в этой книге: сначала нарисовать статичную картинку, потом добавить движение, потом — правила игры, и в последнюю очередь — счёт и «Игра окончена».

Что изучать дальше

ООП подробно, регулярные выражения, веб-разработка и пакеты — четыре двери, открытые этой книгой.

Что изучать дальше

Эта книга — начало пути, а не весь путь. Вот четыре направления, куда можно двигаться дальше, когда мини- и итоговые проекты станут получаться уверенно.

ООП подробно

Глава 15 показала основы классов и объектов. Дальше стоит изучить наследование (когда один класс достраивает другой), магические методы вроде `__str__` и `__eq__`, а также принципы вроде инкапсуляции — они превращают классы из удобной коробки для данных в по-настоящему мощный инструмент проектирования программ.

Регулярные выражения

Модуль `re` позволяет искать и проверять текст по шаблонам — например, «похоже ли это на email-адрес» или «вытащи все числа из строки» — то, что было бы очень неудобно делать вручную через методы строк из главы 8.

Веб-разработка

Глава 22 показала только самые основы Flask. Дальше — базы данных (чтобы данные не исчезали после перезапуска сервера), формы регистрации и входа, полноценные сайты с несколькими страницами и общий фреймворк вроде Django для больших проектов.

Пакеты подробно

Мы уже пользовались пакетами — `pygame`, `flask`. Следующий шаг — научиться собирать *свой собственный* пакет из написанного кода, чтобы переиспользовать его в разных проектах или поделиться им с другими через PyPI.

Что мы узнали в этой книге

- Python — от первой программы print("Привет, мир!") до полноценных игр на Pygame и сайта на Flask.
- Основы языка: переменные, числа и строки, условия и циклы, списки/кортежи/множества/словари.
- Функции и объектно-ориентированное программирование — как упаковывать код в переиспользуемые, понятные блоки.
- Графика и интерфейсы: Turtle, Tkinter и Pygame — три разных

инструмента для трёх разных типов программ.

- Файлы, обработка ошибок и веб-разработка — то, что превращает учебные примеры в программы, готовые к реальному использованию.
- Больше двадцати работающих проектов, каждый из которых можно взять за основу для своей собственной идеи.

Итоги

Спасибо, что прошли этот путь от первой строчки кода до собственных игр и сайта. Дальше — только практика: чем

больше вы пишете, тем увереннее становится каждая следующая программа. Удачи, и до встречи в коде!

Предметный указатель

Быстрый способ найти, в какой главе рассматривается нужный термин.

Термины отсортированы по алфавиту. Номер страницы соответствует канонической вёрстке книги; ссылка ведёт на страницу главы, где термин рассматривается.

Символы и англоязычные термины

А—Я