Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Тверской государственный университет»

С.В. Сорокин

Введение в программирование на языке Python. Практикум

Тверь 2016

УДК 004.42, 519.682

ББК (Ж/О)32.973.26-018.1

Тверской государственный университет

Факультет прикладной математики и кибернетики

Рецензенты:
Доктор технических наук, профессор, А.Н. Райков
Кандидат технических наук, доцент, В.Л. Волушкова

Сорокин С.В. **Введение в программирование на языке Python. Практикум**: учеб. пособие. Тверь: Твер. гос. ун-т, 2015. – 123 с.

Учебное пособие адресовано, студентам, обучающимся по направлению подготовки 010300 «Фундаментальная информатика и информационные технологии». Пособие призвано помочь студентам в организации их самостоятельной работы по изучению программирования на языке Python. Оно содержит ряд практических заданий и рекомендации по их решению. Представленные задания помогут студентам овладеть базовыми конструкциями языка программирования Python и приёмами программирования.

УДК 004.42, 519.682

ББК (Ж/О)32.973.26-018.1

Оглавление

Введение

Учебное пособие разработано для студентов 1-го курса, обучающихся по специальности 010300 «Фундаментальная информатика и информационные технологии».

Пособие призвано помочь студентам в организации самостоятельной работы по изучению программирования на языке Python в рамках дисциплин «Практикум по программированию» и «Практикум на ЭВМ».

Необходимо отметить, что пособие не является самодостаточным учебником для изучения языка Python. Его целью являются дополнение аудиторных занятий и разъяснение того, как рассмотренные на лекциях конструкции языка и приёмы программирования применяются для решения практических задач.

Структура пособия

Пособие структурировано в соответствии с заданиями, решаемыми студентами каждую неделю обучения.

Первое домашнее задание подразумевает установку и настройку программного обеспечения, необходимого для работы.

Последующие задания являются заданиями на программирование. Каждое задание содержит подробное описание постановки задачи и рекомендации по порядку реализации. В некоторых случаях описываются полезные для решения функции или приёмы, которые не рассматривались на занятиях в аудиториях.

Типографские соглашения

Важные понятия выделяются в тексте **полужирным** шрифтом.

Команды языка Python, код программ и результаты их работы выделяются моноширным шрифтом. Имена функций и методов классов в тексте указываются с круглыми скобками, при этом аргументы внутри скобок могут опускаться.

URL-ссылки в тексте выделяются подчёркиванием. Обратите внимание, что неподчёркнутые знаки препинания после ссылки не входят в её состав.

Язык программирования Python

> Программы должны быть написаны для чтения людьми и только по случайности для исполнения машинами.
>
> —Abelson & Sussman, *Structure and Interpretation of Computer Programs*

Язык программирования Python начал разрабатываться в конце 80-х годов XX века голландским программистом Гвидо ван Россумом. Как ни странно, многие удачные программные проекты начинали разрабатывать как хобби – в свободное время и для получения удовольствия, а не официальный проект. Так же обстояли дела и с языком Python – Гвидо выбрал его создание как приятное развлечение на рождественские каникулы в 1989 году. В феврале 1991 года исходный код языка Python был опубликован в группе новостей[1] alt.sources.

Версия 2.0 языка Python была выпушена 16 октября 2000 года. С этого времени разработка языка приняла более открытый и ориентированный на сообщество программистов характер. 3 декабря 2008 года была выпущена версия 3.0, которая устранила некоторые недостатки предыдущих версий. К сожалению, при этом не удалось сохранить полную совместимость со 2-й версией.

Сейчас язык Python является широко распространённым универсальным языком программирования высокого уровня. Язык Python используется в крупных компаниях, работающих в сфере информационных технологий, например, в Google, IBM и Яндекс. На языке Python разрабатывают приложения для Web, приложения для мобильных платформ и обычные приложения с графическим интерфейсом пользователя. Наряду с такими системами, как Matlab, R язык Python часто используют для научных расчётов. При этом его удобно использовать и для того, чтобы быстро написать небольшой скрипт для автоматизации переименования большого числа файлов хитрым образом.

Одной из основных идей, положенных в основу разработки языка Python, является ориентация на простоту языка и хорошую читаемость кода. Практика программирования показывает, что бóльшую часть времени программист

[1] Группа новостей (англ. newsgroup) – хранилище сообщений, отправленных пользователями. Сообщение, отправленное в группу, становится доступным всем её подписчикам. Такие группы использовались для ведения дискуссий в компьютерных сетях до появления форумов и коллективных блогов.

занимается не написанием нового кода, а чтением и анализом существующего – разработанного ранее им, его коллегами или в других группах. Поэтому в настоящее время в профессиональном сообществе хорошая читаемость и соблюдение стиля кода считаются важными достоинствами. Как показала практика, язык Python также стал очень удачным языком для начального обучения программирования.

Стандартная библиотека языка Python включает большой объем полезных функций. Кроме стандартной библиотеки, входящей в базовый комплект языка, существует большое число сторонних библиотек, помогающих разрабатывать приложения для самых разных сфер.

Как и многие другие современные языки программирования, Python поддерживает различные парадигмы программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное. В ходе изучения языка Python вы познакомитесь с этими стилями написания компьютерных программ.

Основная реализация языка Python – CPython. Она имеет **открытую лицензию**. Это означает не только то, что необходимые для программирования на языке Python средства можно скачать и использовать совершенно бесплатно, но и то, что их исходный код также свободно доступен для изучения и модификации. При этом разработанные на языке Python программы могут иметь коммерческую (закрытую) лицензию.

Python – активно развивающийся язык. Новые версии с добавлением или изменением существенных свойств выходят примерно один раз в два с половиной года. Быстрое развитие языка – его преимущество: исправляются ошибки, появляются новые возможности, упрощающие работу программиста. Но в то же время некоторые специалисты отмечают, что такое быстрое развитие языка усложняет его использование в производственных системах – написанный ранее код может оказаться несовместим с предыдущими версиями и его придётся корректировать.

Другим следствием быстрого развития языка является отсутствие официального стандарта языка, утверждённого соответствующей организацией по стандартизации. Фактически роль стандарта языка Python выполняет одна из его реализаций – CPython.

Ситуация с несовместимостью разных версий хорошо иллюстрируется на примере версий 2 и 3 языка Python. Несмотря на то, что 3-я версия была выпущена в 2008 году, даже сейчас, через 6 лет, многие все ещё продолжают использовать 2-ую версию. Это связано с тем, что в некоторых случаях внесённые в третьей версии изменения требуют настолько серьезной переработки кода программ, что программисты предпочитают пользоваться более старой версией, чем полностью переделывать весь свой код. Кроме того, некоторые библиотеки все ещё остаются доступными только для 2-й версии Python, что ещё сильнее затрудняет перенос приложений, использующих такие библиотеки, на новую версию.

Однако данная проблема не является специфичной именно для языка Python. Она является проявлением более общей проблемы – **проблемы унаследованного кода** (англ. legacy code). Программные системы практически никогда не строятся с нуля, а надстраиваются над существующими системами. Первые компьютеры программировались непосредственно в машинных кодах, однако быстро появились первые языки ассемблера, позволившие заменить числовые значения машинных команд на более понятные человеку имена из букв. На языке ассемблера можно написать более сложную программу – например, компилятор[2] такого языка, как C.

Используя универсальный язык программирования, программист не будет задумываться о том, как на конкретной машине реализовывать простейшие арифметические операции для разных видов чисел, сравнения, условия и циклы. Не нужно задумываться над такими вопросами: где должны располагаться операнды, какая конкретно команда процессора должна быть использована для сложения беззнаковых целых чисел размером 2 байта и куда будет помещён результат. Программировать на таком языке проще, быстрее и эффективнее. Кроме того, если язык программирования реализован для разных машин, программу можно будет использовать на любой из них, даже если их архитектура и наборы команд существенно различаются.

Стандартные библиотеки дают программисту готовые средства для выполнения наиболее распространённых операций, таких, как упорядочение последовательностей элементов, действия с текстовыми строками, работа с файлами. Если группа программистов реализует такую операцию, как

[2] Компилятор – программа, преобразующая исходный текст программ в машинный код.

обращение матрицы, и оформит её в виде библиотеки, то другие смогут создать на её основе более сложные вещи – например, библиотеку для создания и обучения искусственных нейронных сетей. На основе этой библиотеки можно будет создать приложение, распознающее автомобильные номера, прогнозирующее погоду на завтра, курс акций или спрос покупателей на товары. Использовать готовые библиотеки быстрее, чем реализовывать всё вновь. Реализация таких операций, как обращение матриц, требует глубоких знаний математики и особенностей выполнения численных операций на компьютерах, которые отсутствуют у многих прикладных программистов.

Использование предыдущих наработок позволяет существенно ускорить разработку программного обеспечения и способствует быстрому прогрессу в области информационных технологий. Но в то же время это делает новые программы зависимыми от огромного объема кода, разработанного ранее и другими людьми: кода операционных систем, компиляторов, стандартных и дополнительных библиотек. Причём каждый из использованных компонентов может зависеть от десятков, и даже сотен других. Создавая новые системы, программисты используют разработанные ранее модули для решения задач в условиях, о которых их разработчики даже не догадывались. В то же время идёт и развитие аппаратного обеспечения, которое даёт дополнительные возможности для программирования. Появляются идеи, технологии и приёмы программирования, которые позволяют повысить эффективность работы.

Такое развитие приводит к тому, что рано или поздно программное обеспечение устаревает: оказывается неудобным и неадекватным для решения новых проблем с использованием новых приемов. Так, например, стандартная библиотека языка C оказалась непригодной для многопоточного программирования. Часто бывает, что выбранные в начале разработки языка или библиотеки решения на практике оказываются не очень удачными. В таких условиях приходится неизбежно расширять возможности языков программирования и их библиотек. При этом перед разработчиками языка возникает дилемма, должна ли новая версия обеспечить работу старых приложений с новой версией языка.

Отказаться от совместимости со старой версией – тяжелое решение. Оно означает, что многие сотни или тысячи программ, написанных на старой версии, придётся переделать. Это означает, что многим программистам придётся заниматься переделкой старых программ вместо создания новых, на

это необходимо потратить много времени и денег. Поэтому во многих случаях стараются сохранить совместимость с предыдущими версиями или, по крайней мере, минимизировать отличия. Так обычно поступают, например, разработчики языков C и C++. Но этот подход не идеален – поддержка совместимости часто не позволяет возможности внести принципиальные изменения в механизмы языка, создаёт массу особенностей в реализации тех или иных операций, которые присутствуют только для обеспечения совместимости со старыми версиями. Такие особенности усложняют жизнь как разработчиков, так и пользователей языка программирования. Во многих случаях приходится сохранять как штатные особенности работы, так и ошибки[3].

При переходе от второй к третьей версии языка Python его разработчики решили пойти другим путём – они внесли в него изменения, которые являются несовместимыми со второй версией. Прежде всего эти изменения коснулись поддержки работы с многоязыковыми текстами, представления базовых типов данных и механизма объектно-ориентированного программирования. Это привело к тому, что существенная часть библиотек и программ, разработанных ранее на языке Python, не была перенесена на новую версию, а продолжает использовать старую. В настоящее время разработчики языка Python в первую очередь выпускают обновления для третьей версии, по возможности[4] и с некоторой задержкой – для второй. При разработке новых программ на языке Python рекомендуют использовать третью версию, кроме случая, когда необходимо воспользоваться библиотекой, доступной только для второй версии.

В данном курсе будет рассмотрена третья версия языка Python. Отметим, что в рамках курса разница между второй и третьей версиями незначительна. Наиболее существенными отличиями являются возможность появления проблем с текстом на русском языке при переходе на вторую версию с третьей и разный результат выполнения операций деления целых чисел.

---

[3] Например, программа Microsoft Excel до сих пор некорректно считает 1900 год високосным [22], так как при разработке первой версии Excel было решено обеспечить полную совместимость с программой Lotus 1-2-3, разработанной фирмой Lotus (куплена IBM) и выпущенной в 1983 году. А версия Excel под Mac не распознаёт даты до 1 января 1904 года в связи с совместимостью с ранними компьютерами Apple Macintosh. Данные особенности учитываются не всеми библиотеками для работы с файлами Excel, что до сих пор приводит к ошибкам при работе с этими файлами в других программах [21].

[4] Внесение некоторых изменений из третьей версии во вторую признаётся слишком трудоёмким и не выполняется.

Полезные ресурсы

Официальным сайтом языка Python и его реализации CPython является сайт https://www.python.org/. На этом сайте можно найти реализацию CPython, документацию по языку, ссылки на другие сайты с информацией о Python.

Документация на третью версию языка Python находится по URL-адресу https://docs.python.org/3/ и включает:

- Language Reference (https://docs.python.org/3/reference/index.html) – описание языка Python;
- Library Reference (https://docs.python.org/3/library/index.html) – описание стандартной библиотеки;
- Tutorial (https://docs.python.org/3/tutorial/index.html) – введение в программирование на языке Python;
- Python Setup and Usage (https://docs.python.org/3/using/index.html) – руководство по установке и использованию Python;
- Installing Python Modules (https://docs.python.org/3/installing/index.html) – описывает установку дополнительных модулей для Python;
- Python HOWTOs (https://docs.python.org/3/howto/index.html) – подробная документация по некоторым распространённым задачам;
- FAQs (https://docs.python.org/3/faq/index.html) – часто задаваемые вопросы и ответы на них.

Официальная документация на Python написана на английском языке. Список русскоязычных ресурсов по Python находится по адресу https://wiki.python.org/moin/RussianLanguage.

Обновления языка Python и другая информация о языке и его окружении публикуются в виде документов под названием **Python Enhancement Proposals**, или сокращенно **PEP**. Список всех PEP находится по адресу http://legacy.python.org/dev/peps/.

Одним из важных PEP является PEP-8 [1], в котором описан рекомендованный стиль оформления кода на языке Python. Следование стандартному стилю оформления является важным качеством для профессионального программиста. Один из русских переводов PEP-8 доступен по адресу http://pep8.ru/doc/pep8/.

По языку Python издано много книг и учебников, среди которых можно отметить [2], [3] и [4].

Средства программирования для Python

Интерпретатор Python

В своей стандартной реализации язык Python является **интерпретируемым** языком программирования. CPython относится к **интерпретаторам**, которые в начале переводят текст программы в промежуточный код, а затем выполняют его в виртуальной машине.

Интерпретатор языка Python запускается командой «`python`». В нём можно набирать команды языка, которые будут сразу же выполняться. Такой режим работы называется **интерактивным**. Интерактивный режим удобен, чтобы быстро проверить, будет ли работать какой-то фрагмент программы.

Среда разработки

Для написания всего текста программы интерактивный режим не используют. Для этого часто используют специальные программы, называемые **интегрированная среда разработки** (англ. Integrated Development Environment, IDE). Такие среды включают:

- специализированный текстовый редактор с функциями подсветки синтаксиса, автодополнения текста и навигации по проекту;
- компилятор и/или интерпретатор;
- средства для управления файлами исходных кодов;
- **отладчик** – средство для поиска ошибок в исходном коде, он позволяет просматривать состояние программы во время её выполнения.

В состав базового дистрибутива Python включена простая среда разработки – IDLE. Она обеспечивает все основные возможности, характерные для таких сред, и считается подходящей для начинающих программистов.

Существуют и другие среды разработки для Python. Среди них можно назвать PyCharm [5], PyScripter [6], Spyder [7]. Они гораздо более удобны и обеспечивают больше функций, чем IDLE.

Альтернативные реализации языка Python

Кроме CPython существуют и другие реализации языка Python. Можно отметить следующие версии:

- IronPython – реализация языка Python для платформы .NET фирмы Microsoft;
- Jython – реализация языка Python, работающая на виртуальной машине Java;
- PyPy – быстрая реализация языка Python;
- Stackless Python – реализация Python с поддержкой многопоточности.

Кроме альтернативных реализаций самого языка есть и модифицированные комплекты, включающие CPython с набором дополнительных библиотек, ориентированных на решение определённых задач. К этой группе дистрибутивов Python относятся:

- Anaconda Python – ориентирован на управление и анализ данных, визуализацию больших наборов данных;
- Conceptive Python SDK – ориентирован на бизнес-приложения и работу с базами данных;
- Portable Python – среда программирования Python с дополнительными библиотеками, настроенная для запуска в Windows со внешнего носителя без установки;
- WinPython – среда разработки Python для научных расчётов, которая может использоваться как в портативном, так и обычном режимах.

Такие дистрибутивы упрощают установку Python с дополнительными библиотеками для использования в конкретных сферах, а портативные дистрибутивы (такие, как Portable Python и WinPython) позволяют быстро запустить и использовать Python в случае, если пользователь не имеет прав для установки программного обеспечения на компьютере.

1. Установка необходимого программного обеспечения

Для выполнения первых упражнений потребуется установить на компьютер интерпретатор Python и две библиотеки – Pygame и SimpleGUICS2Pygame. Рекомендуется также установить утилиту PEP8, предназначенную для проверки исходного кода на соответствие правилам оформления.

Установка интерпретатора Python

Для того чтобы писать программы на языке Python, необходимо установить на компьютер одну из его реализаций. Основной реализацией является CPython, официальный сайт которой находится по URL-адресу https://www.python.org/. Эта реализация доступна для многих платформ, включая Windows, Mac OS X и большинство Unix-совместимых систем.

Во многих Unix-системах (и в Max OS) Python может быть установлен по умолчанию. Прежде чем заниматься установкой, стоит проверить присутствие уже установленных версий, набрав команду «python» в командной строке. Если в системе установлена одна из версий Python, то будет выведено приглашение интерпретатора, включающее номер установленной версии.

```
c:\>python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

В курсе будет использоваться язык Python версии 3.4. Можно использовать и другие версии. При наличии любой из 3-х версий вы скорее всего не столкнётесь с какими-либо заметными отличиями, кроме установки специфичной версии библиотеки Pygame. Среду Python второй версии лучше сразу обновить на новую.

Для установки среды Python в операционной системе Windows следует скачать программу установки с URL-адреса https://www.python.org/download/. На этой странице находятся ссылки для скачивания последних модификаций второй и третьей версий языка Python для разных систем. Выберите подходящую систему (32 или 64 бит) и загрузите файл «MSI Installer». После скачивания запустите файл и следуйте указаниям программы установки.

Если требуется установить более новую версию Python в Unix-системах, то, как правило, следует воспользоваться стандартным репозитарием.

Установка библиотек и дополнительных утилит

На сайте Python поддерживается список дополнительных пакетов (библиотек, утилит, средств разработки и других программ), который называется **Python Package Index (PyPI)**. Он доступен по адресу https://pypi.python.org/pypi. Каталог PyPI содержит более 48 тысяч пакетов. Установить любой из них можно с помощью команды pip, находящейся в поддиректории Scripts директории Python[5]. Для установки пакета достаточно выполнить команду «`pip install имя_пакета`». Следующий пример показывает, как проходит установка утилиты `pep8`:

```
c:\Python34\Scripts>pip install pep8
Downloading/unpacking pep8
  Downloading pep8-1.5.7-py2.py3-none-any.whl
Installing collected packages: pep8
Successfully installed pep8
Cleaning up...
```

Для удаления пакета достаточно выполнить команду «`pip uninstall имя_пакета`»[6]:

```
c:\Python34\Scripts>pip uninstall pep8
Uninstalling pep8:
  c:\python34\lib\site-packages\__pycache__\pep8.cpython-34.pyc
  c:\python34\lib\site-packages\pep8-1.5.7.dist-info\description.rst
  c:\python34\lib\site-packages\pep8-1.5.7.dist-info\entry_points.txt
  c:\python34\lib\site-packages\pep8-1.5.7.dist-info\metadata
  c:\python34\lib\site-packages\pep8-1.5.7.dist-
info\namespace_packages.txt
  c:\python34\lib\site-packages\pep8-1.5.7.dist-info\pydist.json
  c:\python34\lib\site-packages\pep8-1.5.7.dist-info\record
  c:\python34\lib\site-packages\pep8-1.5.7.dist-info\top_level.txt
  c:\python34\lib\site-packages\pep8-1.5.7.dist-info\wheel
  c:\python34\lib\site-packages\pep8.py
  c:\python34\scripts\pep8.exe
Proceed (y/n)? y
  Successfully uninstalled pep8
```

[5] До версии 3.4 Python не включал pip в стандартном дистрибутиве, и его надо было устанавливать дополнительно.

[6] Удаление пакета pep8 показано только для примера. Рекомендуется оставить его установленным и использовать для проверки разрабатываемых программ.

Установка библиотеки Pygame

Даже PyPI не включает все существующие библиотеки для Python. Библиотеки, не включенные в PyPI, необходимо скачивать и устанавливать самостоятельно. К таким библиотекам относится и используемая в курсе библиотека Pygame.

Для установки библиотеки Pygame выполните следующие шаги:

1) скачайте со страницы http://www.lfd.uci.edu/~gohlke/pythonlibs/ файл:
   a. pygame-1.9.2a0.win-amd64-py3.4.exe если у Вас 64-битный Windows;
   b. pygame-1.9.2a0.win32-py3.4.exe если у Вас 32-битный Windows;
2) запустите скачанный файл;
3) нажмите «Далее>»;
4) убедитесь, что установщик правильно нашел путь к Python;
5) нажимайте «Далее>» до завершения установки.

Версия Pygame должна соответствовать установленной версии Python (файлы, оканчивающиеся на …py3.4 предназначены для Python 3.4). Библиотека Pygame не сможет работать в другой версии Python (установщик сообщит, что необходимая версия не найдена). Если Вы хотите использовать другую версию Python, найдите подходящий Pygame.

Установка библиотеки SimpleGUICS2Pygame

Эта библиотека входит в Python Package Index (PyPI) и может быть установлена с помощью менеджера пакетов pip. Для установки достаточно выполнить команду `pip install SimpleGUICS2Pygame`.

Установка должна выглядеть следующим образом:

```
c:\Python34\Scripts>pip install SimpleGUICS2Pygame
Downloading/unpacking SimpleGUICS2Pygame
  Running setup.py
(path:C:\Users\sergey\AppData\Local\Temp\pip_build_sergey\SimpleGUICS2Pyg
ame\setup.py) egg_info for package SimpleGUICS2Pygame

Installing collected packages: SimpleGUICS2Pygame
  Running setup.py install for SimpleGUICS2Pygame

Successfully installed SimpleGUICS2Pygame
Cleaning up...
```

Если не удалось установить SimpleGUICS2Pygame или pygame

Для решения задач можно использовать онлайн версию Python, которая находится по адресу http://www.codeskulptor.org/.

Учтите, что в системе CodeSculptor используется 2-я версия Python, поэтому, скорее всего, не будет правильно работать вывод текста на русском языке, используйте английский язык.

Далее приведены другие отличия, которые могут встретиться при использовании системы CodeSculptor вместо рассматриваемой на занятиях среды программирования на основе 3-ей версии языка Python (этот список не является исчерпывающим).

1. В CodeSculptor для подключения библиотеки SimpleGUI необходимо в начале программы вместо
   ```
   import SimpleGUICS2Pygame.simpleguics2pygame as simplegui
   ```
   писать
   ```
   import simplegui
   ```
2. Для печати в консоль вместо
   ```
   print(a)
   ```
   напишите
   ```
   print a
   ```
   (без круглых скобок).
3. Оператор деления / по-разному работает для целочисленных аргументов.

PEP8

Как уже было изложено выше, утилита pep8 используется для проверки оформления текста программы стилистическим требованиям, изложенным в рекомендации PEP-8.

Для её запуска необходимо набрать в командной строке имя pep8, указать флаги (если требуется) и затем перечислить файлы для проверки. Ниже приведён пример проверки программы optparse.py с флагом --first – выводить только первое место каждой ошибки.

```
$ pep8 --first optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
```

```
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

Флаг `--show-source` предписывает выводить место в программе, в котором была обнаружена ошибка, а `--show-pep8` – соответствующий текст из PEP8:

```
$ pep8 --show-source --show-pep8 testsuite/E40.py
testsuite/E40.py:2:10: E401 multiple imports on one line
import os, sys
         ^
    Imports should usually be on separate lines.

    Okay: import os\nimport sys
    E401: import sys, os
```

С другими возможностями pep8 можно ознакомиться, запустив её с флагом `--help`.

2. Камень, ножницы, бумага, ящерица, Спок

Постановка задачи

«Камень, ножницы, бумага» – популярная детская игра. В этом задании Вам нужно разработать программу, с которой можно играть в расширенный вариант этой игры, известной как «камень, ножницы, бумага, ящерица, Спок». Правила расширенного варианта следующие:

Ножницы режут бумагу. Бумага заворачивает камень. Камень давит ящерицу, а ящерица травит Спока, в то время как Спок ломает ножницы, которые, в свою очередь, отрезают голову ящерице, которая ест бумагу, на которой улики против Спока. Спок испаряет камень, а камень, разумеется, затупляет ножницы.

Графически, они представлены на рисунке 2.1 (стрелки показывают, кто кого побеждает).

Рис. 2.1. По часовой стрелке с верхнего: ножницы, бумага, камень, ящерица, Спок

В задании необходимо реализовать функцию `rpsls(name)`, которая получает на вход строку name, содержащую выбор пользователя, случайным образом делает свой выбор и печатает результат игры.

Если попробовать закодировать правило выигрыша напрямую, используя конструкцию `if/elif/else`, то потребуется 25 условных выражений. Однако есть более простой способ. Если закодировать каждый выбор следующим образом:

- камень – 0,
- Спок – 1,

- бумага – 2,
- ящерица – 3,
- ножницы – 4,

то каждый вариант выигрывает у двух предыдущих и проигрывает двум следующим за ним (если использовать арифметику по модулю).

Рекомендованный порядок выполнения задания

Для решения задания используйте прилагаемый файл с шаблоном rspls_template.py.

1. Реализуйте вспомогательную функцию `name_to_number(name)`, которая преобразует строку `name` в число в диапазоне от 0 до 4, как описано выше. Эта функция должна использовать конструкцию `if/elif/else`. Используйте условия вида `name == 'бумага'`, чтобы различить случаи. Для упрощения отладки добавьте в конце условного оператора раздел `else` для случая, когда `name` не соответствует ни одной из правильных входных строк, и распечатайте сообщение об ошибке. Функцию `name_to_number()` можно проверить, используя шаблон name_to_number_test.py.

2. Реализуйте вспомогательную функцию, `number_to_name(number)`, которая преобразует число в диапазоне от 0 до 4 в строку с соответствующим названием. В этой функции тоже можно использовать конструкцию `if/elif/else` с печатью сообщения об ошибке в разделе else.
   Проверьте эту функцию с помощью шаблона number_to_name_test.py.

3. Реализуйте первую часть основной функции `rpsls(player_choice)`. Распечатайте пустую строку (чтобы разделить последовательные игры), затем – строку с сообщением о выборе игрока (переданном в параметре `player_choice`). Затем вычислите значение `player_number`, соответствующее выбору игорка, вызвав функцию `name_to_number()` с аргументом `player_choice`.
   Проверьте, что первая часть функции работает, с помощью вызовов в конце шаблона.

4. Реализуйте вторую часть функции `rpsls()`, которая генерирует выбор компьютера и печатает соответствующее сообщение. Используйте функцию `random.randrange()` чтобы сгенерировать случайное значение `comp_number` между 0 и 4, которое соответствует выбору компьютера. Поэкспериментируйте с `randrange()` в интерактивном

режиме, чтобы лучше понять, как она работает и как генерировать числа в правильном диапазоне. После этого вычислите строку `comp_name`, соответствующую выбору компьютера, используя функцию `number_to_name()`. Напечатайте соответствующее сообщение. Проверьте, что добавленная часть функции работает.

5. Реализуйте последнюю часть функции `rpsls()` которая определяет и печатает сообщение. Для этого вычислите разницу между `comp_number` и `player_number` по модулю 5. Затем напишите оператор `if/elif/else`, который проверит различные возможные значения разности и напечатает соответствующее сообщение о победителе.

Проверьте, что функция работает. Так как мы ещё не познакомились со средствами, которые позволяют получить ввод от пользователя, просто вызывайте Вашу функцию, передавая ей текстовые строки со своим выбором.

Функция random.randrange()

Модуль `random` в стандартной библиотеке языка Python содержит функции для генерации случайных чисел. Функция `randrange()` генерирует и возвращает случайное число из заданного диапазона:

- вызов `randrange(stop)` даст число из диапазона от 0 до `stop`-1 включительно;
- `randrange(start,stop)` – число из диапазона от `start` до `stop`-1 включительно;
- `randrange(start,stop,step)` – числа из диапазона `start` до `stop`-1 включительно с шагом `step`;
- если `stop<start`, а `step` отрицателен, то числа в диапазоне идут в сторону уменьшения.

Примеры диапазонов:

```
randrange(10) – случайное число из {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
randrange(1,11) – случайное число из {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
randrange(0,10,3) – случайное число из {0, 3, 6, 9}
randrange(0,-10,-1) – случайное число из {0, -1, -2, -3, -4, -5, -6, -7,
-8, -9}
```

Пример использования `randrange()`:

```
import random
print(random.randrange(10), random.randrange(1,11), random.randrange(0,10,3))
```

Этот код напечатает 3 случайных числа. Первое число из диапазона $[0, \dots, 9]$, второе – из $[1, \dots, 10]$, а третье – из набора $[0, 3, 6, 9]$. Ниже показаны возможные варианты вывода (при каждом запуске значения будут разными):

```
>>> import random
>>> print(random.randrange(10),random.randrange(1,11),random.randrange(0,10,3))
3 9 0
>>> print(random.randrange(10),random.randrange(1,11),random.randrange(0,10,3))
0 2 6
>>> print(random.randrange(10),random.randrange(1,11),random.randrange(0,10,3))
7 8 0
>>> print(random.randrange(10),random.randrange(1,11),random.randrange(0,10,3))
2 7 9
```

3. Угадай число

Постановка задачи

Угадай число – одна из простейших игр для двух игроков. Один игрок задумывает число из известного диапазона, второй пытается угадать это число. После каждой попытки первый игрок отвечает «больше», «меньше» или «угадал» в зависимости от того, является задуманное им число большим, меньшим или равным предложенному. В этом задании Вам надо реализовать программу на Python, в которой компьютер будет загадывать числа, а пользователь – угадывать.

Взаимодействие с программой будет организовано с помощью поля ввода и нескольких кнопок. В этом задании не требуется использовать холст, результаты игры будут печататься в консоли[7].

Рекомендованный порядок выполнения задания

В качестве основы используйте шаблон numbers_template.py. Рекомендуется решать задачу в следующем порядке.

1. Определите, какие глобальные переменные потребуются для хранения состояния игры. Например, достаточно очевидным представляется хранить число, «задуманное» (сгенерированное случайным образом) программой. Потребуются и другие переменные, особенно для реализации дополнительных расширений.
2. Вспомните, как в языке Python сгенерировать случайное число в заданном диапазоне от `low` до `high`. При обсуждении диапазонов будем следовать принятому в Python соглашению, что числа диапазона включают нижнюю границу и не включают верхнюю, что в математической записи выражается как $[low, high)$. Таким образом, $[0,3)$ включает числа 0, 1 и 2. Начните свою реализацию с работы в диапазоне $[0,100)$.
3. Вспомните, как создать поле ввода текста, используя модуль `simplegui`. Это поле будет использоваться для ввода предположения пользователя.

[7] Сначала реализовать программу так, чтобы она печатала данные в консоли, а затем реализовать графический интерфейс – хорошая тактика и для решения задач в дальнейшем. Это позволяет вначале сфокусироваться на логике программы и правильно её реализовать, а уже затем задумываться о том, как её красиво представить. Поиск ошибок в программе, когда она работает в режиме графического интерфейса, часто оказывается сложным.

4. Напишите функцию-обработчик `input_guess(guess)`, которая принимает предположение (англ. guess) пользователя, сравнивает с задуманным числом и печатает соответствующее сообщение. Подсказка: потребуется преобразовать предположение пользователя из строки в число.
5. Напишите код для создания окна, поля ввода и запуска окна.
6. Проверьте Ваш код, запуская игру несколько раз. На этом этапе после завершения игры программу надо закрыть и запустить заново. Вы также можете использовать шаблон numbers_test.py для тестирования.
7. Напишите функцию new_game(), которая должна начинать новую игру. Внутрь этой функции поместите генерацию случайного числа. Поместите вызов этой функции перед созданием окна.

На этом этапе Вы получите минимальную по возможностям программу, которая может быть сдана. Дальнейшие доработки помогут Вам получить дополнительные баллы.

Дополнительные задания

1. Добавьте две кнопки для перезапуска игры с разными диапазонами. Это должны быть кнопки «Диапазон: $0-100$» и «Диапазон: $0-1000$». Нажатие на любую из этих кнопок должно приводить к перезапуску игры и печати соответствующего сообщения. Они должны работать в любой момент во время игры.
   При запуске игры она должна сразу же начинать игру в диапазоне $0-100$. Когда игра завершилась, она должна сразу же перезапуститься в том же диапазоне, в котором проходила.
   Подсказка: при реализации этого пункта может быть полезным ввести глобальную переменную.

Играя в «Угадай число», можно заметить, что хорошей стратегией является запоминание диапазона, состоящего из максимального предположения, которое «меньше» задуманного числа, и минимального, которое «больше». Хорошим кандидатом на следующее предположение будет среднее арифметическое между этими двумя числами. Ответ на это предположение позволит определить следующий интервал, содержащий задуманное число, и он будет в два раза меньше исходного. Например, если игра проходит в диапазоне $[0,100)$, можно начать с предположения 50. Если ответом будет «больше», значит задумано число в диапазоне $[51,100)$. На

следующем шаге можно предложить 75 и так далее. Такой способ сокращения пространства поиска в программировании известен как двоичный поиск [8].

Следуя идеи двоичного поиска, можно угадать любое число в диапазоне $[low, high)$ не более чем за n шагов, где n – минимальное число, такое что $2^n \geq high - low + 1$. Для диапазона $[0{,}100)$ $n = 7$, для диапазона $[0{,}1000)$ $n = 10$.

2. Вторым добавлением к «Угадай число» будет ограничение числа попыток, за которые игрок должен угадать число. После каждой попытки игра должна печатать число оставшихся попыток. Если игрок использовал все попытки, то он проигрывает, и игра должна распечатать соответствующее сообщение.
   В качестве ограничений используйте числа, указанные выше. Подумайте, как можно вычислить соответствующее число по заданным границам диапазона, используя функции `math.log()` и `math.ceil()`.

Оценка задания

Игра без дополнительных расширений – 60 %.

Реализация кнопок для выбора диапазона – 20 %.

Реализация ограничения на число попыток – 20 %.

4. Секундомер

Постановка задачи

На этой неделе потребуется разработать простой цифровой секундомер, отслеживающий время с точностью до одной десятой доли секунды. Секундомер должен управляться кнопками «Старт», «Стоп» и «Сброс». Для создания секундомера потребуется совместить работу с таймером, форматированием строк и выводом текста на холст. Для упрощения работы рекомендуется использовать шаблон stopwatch_template.py.

Рекомендованный порядок работы над заданием

1. Создайте таймер с интервалом срабатывания 0.1 секунды, обработчик которого будет увеличивать целое число, хранящееся в глобальной переменной. Это число будет отслеживать время. Не забывайте, что аргумент функции `create_timer()` – время в миллисекундах (1 мс – одна тысячная секунды).
   Проверьте, что таймер работает правильно, печатая значение переменной времени в консоли.
   Не используйте для хранения времени числа с плавающей точкой. Скорее всего такой вариант можно заставить работать, но только с приложением больших усилий на борьбу с погрешностями. Храните время с помощью целого числа, например $12 = 1.2$ секунды.

2. Напишите обработчик рисования, который отобразит время в виде числа в центре холста. Пока не задумывайтесь о красивом форматировании, просто используйте функцию `str()`, чтобы преобразовать число в строку и вывести как есть.

3. Добавьте кнопки «Старт» и «Стоп», обработчики которых будут запускать и останавливать таймер.
   Затем добавьте кнопку «Сброс», которая останавливает таймер и сбрасывает время в 0.
   При запуске программы секундомер должен быть остановлен.

4. Напишите вспомогательную функцию `format(t)`, которая должна возвращать строку в формате $A{:}\,BC.D$, где A – минуты, BC – секунды (B – цифра от 0 до 5), D – десятые доли секунд.
   Проверьте функцию `format(t)` с помощью теста в файле

format_template.py. Обращайте внимание, чтобы возвращаемая функцией строка всегда содержала необходимые ведущие нули:

- `format(0) = '0:00.0'`
- `format(11) = '0:01.1'`
- `format(321) = '0:32.1'`
- `format(613) = '1:01.3'`

Подсказка: используйте операторы «целочисленное деление» (`//`) и «остаток от деления» (`%`) чтобы получить значения A,B,C и D из исходного числа.

5. Добавьте вызов функции `format()` в обработчик рисования, чтобы получить законченный секундомер.

Дополнительные задания

1. Превратите секундомер в тренажер реакции. Для этого добавьте к секундомеру два счётчика, которые будут отслеживать, сколько раз секундомер был остановлен и сколько из них удалось его остановить в ровную секунду (1.0, 2.0, 3.0 и т.д.). Значение этих счётчиков должно отображаться в верхней правой части холста в формате "x/y", где y – общее число остановок секундомера, x – число остановок в ровную секунду.

2. Добавьте код, который не позволит увеличивать счётчики нажатием на кнопку «Стоп», если секундомер уже остановлен.
   Подсказка: для этого может быть удобно добавить булеву переменную, которая будет иметь значение True, если секундомер запущен, и False – если остановлен.

3. Модифицируйте обработчик кнопки «Сброс» так, чтобы он сбрасывал значения счётчиков в 0 при сбросе секундомера.

Оценка задания

Полностью работоспособный секундомер с функциональностью, описанной в разделе «Рекомендованный порядок выполнения задания» – 70 %.

Форматирование времени реализовано некорректно – -30 %.

Добавлены счётчики нажатий из «дополнительных расширений» – 15 %.

Кнопки «Стоп» и «Сброс» корректно работают с подсчётом нажатий – 15 %.

5. Пинг-понг

Постановка задачи

В этом домашнем задании требуется разработать версию игры «Пинг-понг» – одной из первых аркадных видео-игр (1972г.) [9]. Реализация этой игры позволит развить необходимые навыки для разработки игры «Астероиды», задание по которой вы получите позднее.

Чтобы получить представление о создаваемой игре, можно воспользоваться одной из её реализаций в web, например: http://libcanvas.github.io/games/pingpong/ (управление клавишами w-s и вверх-вниз). Ваша реализация будет в некоторых деталях отличаться от этой.

Для упрощения работы рекомендуется использовать шаблон pong_template.py.

Рекомендованный порядок работы над заданием

1. Добавьте код к шаблону программы, которая рисует мяч, движущийся по пинг-понг столу. Рекомендуется добавить обновление позиции мяча в обработчик рисования, как в примерах ball.py, velocity_control.py.

2. Добавьте код в функцию `spawn_ball()`, которая помещает мяч в середину игрального стола и присваивает мячу фиксированную скорость по Вашему выбору (на данный момент). Игнорируйте параметр `direction` на данном этапе.

3. Добавьте вызов функции `spawn_ball()` в функцию `new_game()`, которая запускает игру Пинг-понг. Обратите внимание, что шаблон программы включает первоначальный вызов `new_game()` в основной части вашей программы, чтобы запустить игру.

4. Измените существующий код так, чтобы при столкновении с нижней и верхней стенками мяч отскакивал от них. Протестируйте код с различными начальными скоростями мяча.

5. Добавьте случайный выбор скорости в `spawn_ball(direction)`. Скорость мяча должна быть направлена вверх и вправо, если `direction == RIGHT`, и вверх и влево, если `direction == LEFT`.
   Значения для горизонтальных и вертикальных компонентов скорости должны быть получены с использованием `random.randrange()`. Предполагается, что для горизонтальной скорости хорошо использовать

`random.randrange(120, 240)` пикселей в секунду, для вертикальной скорости – `random.randrange(60, 180)` пикселей в секунду.

6. Добавьте в обработчика рисования код для проверки касания мяча с левым и правым бортами стола. Помните, что борта смещены от левого и правого краёв холста на ширину ракетки. При касании мячом бортика, используйте функции `spawn_ball(LEFT)` и `spawn_ball(RIGHT)`, чтобы установить мяч в центре части стола так, чтобы он двигался к противоположному борту.

7. Далее добавьте код, который рисует левую и правую ракетки соответствующих сторон. Вертикальные позиции обеих ракеток должны храниться в двух глобальных переменных. В шаблоне использовались переменные `paddle1_pos` и `paddle2_pos` (от англ. `paddle` – `ракетка`).
   Подсказка: для рисования ракетки можно использовать функцию `canvas.draw_line()`, задав большую толщину линии:

```
                координаты одного конца;  координаты второго конца;
                                ↓                 ↓
canvas.draw_line(  [10,20],  [30,40],  12,    'Red')
                                                ↑        ↑
                                          толщина линии;  цвет линии.
```

8. Добавьте код, который изменяет значения вертикальных позиций ракеток, в обработчик рисования. Новая позиция ракеток должна зависеть от двух глобальных переменных, отвечающих за вертикальные скорости ракеток. В шаблоне такими переменными являются `paddle1_vel` и `paddle2_vel`.

9. Для обновления значений обеих вертикальных скоростей используйте обработчики клавиш. Клавиши `"w"` и `"s"` изменяют вертикальную скорость левой ракетки, клавиши `"стрелка вверх"` и `"стрелка вниз"` – скорость правой ракетки. Левая ракетка движется вверх и вниз с постоянной скоростью при нажатии `"w"` и `"s"` соответственно и остаётся неподвижной, если ни одна из клавиш не нажата. Для достижения такого результата следует использовать обработчики `KeyDown()` и `KeyUp()`, чтобы соответствующим образом изменять вертикальную скорость.

10. Чтобы ракетки не выходили за край холста, следует ограничить их движение, добавив проверку перед обновлением вертикальных позиций

ракетки в обработчике рисования. В частности, следует проверить, не выходит ли за край экрана часть ракетки из-за текущего обновления её позиции. Если часть ракетки не видна, не производите данное обновление.

11. Измените код, контролирующий столкновение с левым и правым бортами (шаг 6) таким образом, чтобы осуществлялась проверка, что мяч ударился об ракетку, когда он касался бортика. Если это так, отразите мяч обратно на поле. Такая модель столкновения исключает возможность мяча попасть в ребро ракетки и значительно упрощает код, обрабатывающий столкновение/отражение.
12. Для усложнения игры увеличивайте скорость мяча на 10% каждый раз при отбивании ракеткой.
13. Добавьте ведение счета игры. Каждый раз, когда мяч попадает в левый или правый бортик и не попадает в ракетку, игрок напротив получает очко и мяч помещается в центр стола для разыгрывания.
14. Добавьте код в функцию `new_game()` для обнуления счёта перед вызовом функции `spawn_ball()` и добавьте кнопку «Перезапустить игру», которая вызывает `new_game()` для сброса счет и разыгрывания мяч.

Окончательный вариант игры достаточно сильно похож на оригинал аркадной игры Пинг-понг. Реализация игры на Python может занять чуть больше 100 строк кода с комментариями.

Оценка задания

Мяч корректно запускается в начале игры – 10 %.

Мяч корректно отражается от верхней и нижней стенок – 10 %.

Учитывается попадание в левый и правый край стола (а не в край холста), мяч перезапускается в сторону игрока, выигравшего последнее очко – 10 %.

Корректная работа с ракетками:

- ракетки управляются клавишами w/s и вверх/вниз – 20 %;
- ракетки рисуются вровень с краями поля и не выходят вверх/вниз – 20 %;
- мяч отражается от ракеток – 20 %.

Подсчёт очков и перезапуск игры кнопкой – 10%.

6. Мемори

Постановка задачи

В этом задании сделаем игру на тренировку памяти, известную под названиями «Мемори» или «Угадай пару». Игра должна проходить по следующим правилам.

- Играет один игрок.
- Перед началом игры перед игроком находятся 16 перевёрнутых карточек с цифрами от 0 до 7, по две карточки с каждой цифрой.
- Игрок делает ход: выбирает щелчком мыши и переворачивает две карточки. Если на этих карточках написаны одинаковые цифры, то карточки и дальше останутся открытыми. Если на карточках написаны разные цифры, то после того, как игрок выберет следующую карточку, выбранная ранее пара карточек закроется.
- Задача игрока: открыть все карточки за минимальное число ходов.
- При запуске программы игра должна начаться без дополнительного нажатия на кнопку «Перезапустить игру».
- После того, как игра закончится, игрок должен нажать на кнопку «Перезапустить игру» для её начала.

Для упрощения работы рекомендуется использовать шаблон memory_template.py.

Пример игры

Начало игры

Все карты скрыты.

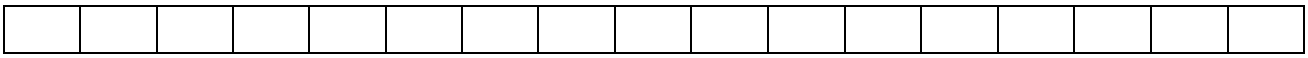


Рис. 6.1. Экран в начале игры

Первый ход

Игрок выбирает одну карту, щелкает по ней мышью, она открывается.

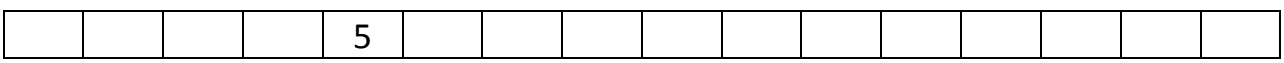


Рис. 6.2. Экран игры в начале первого хода

Игрок выбирает вторую карту, щелкает по ней, она тоже открывается.

				5					3						

Рис. 6.3. Экран игры в конце первого хода

Карты разные, в начале следующего хода они будут закрыты.

Второй ход

Игрок выбирает ещё одну карту. Выбранная им карта открывается, непарные карты, открытые на предыдущем ходу закрываются.

		4													

Рис. 6.4. Экран игры в начале второго хода

Игрок выбирает следующую карту.

		4				4									

Рис. 6.5. Экран игры в конце второго хода

Карты совпали, поэтому дальше они останутся открытыми.

Третий ход

Игрок выбирает карту, она открывается, предыдущие открытые карты остаются открытыми.

		4				4						7			

Рис. 6.6. Экран игры в начале третьего хода

Игрок выбирает вторую карту, она открывается.

		4				4	6					7			

Рис. 6.7. Экран игры в конце третьего хода

Карты не совпали, они будут закрыты.

Четвёртый ход

Игрок выбирает карту, выбранная карта открывается, карты с третьего хода закрываются.

		4				4								3	

Рис. 6.8. Экран игры в начале четвёртого хода

Игрок выбирает ещё одну карту.

		4				4			3					3	

Рис. 6.9. Экран игры в конце четвёртого хода

Найдена вторая пара карт. Эти карты также останутся открытыми до конца игры.

Конец игры

Игра кончается, когда игрок откроет все карты.

1	2	4	2	5	6	4	6	1	3	8	8	7	9	3	9

Рис. 6.10. Экран в конце игры

Наблюдения за ходом игры

Обратите внимание на следующие моменты:

- Один ход игры включает **два** щелчка мышью по **закрытым** картам. Подсчёт ходов должен вестись именно таким образом.
- Щелчки по открытым картам не должны приводить ни к каким изменениям в состоянии игры.
- Карты открываются только по выбору пользователя: одна карта по щелчку мышью.
- Закрыться могут только те две карты, которые были открыты на предыдущем ходе. Карты, открывавшиеся больше одного хода назад, не закрываются.

Рекомендованный порядок работы над заданием

1. В игре «Мемори» используется набор из 16 карточек с цифрами в интервале $[0,8)$, каждая цифра встречается дважды. Предполагается, что для создания полного списка карточек вы произведёте конкатенацию двух списков с цифрами от 0 до 8. Такой список можно сделать руками или с помощью функции `range()`.

2. Напишите обработчик рисования, который перебирает игровые карточки, используя цикл `for`, и рисует карточки с цифрами на холсте, используя `draw_text()`.

3. Перемешайте карточки, используя `random.shuffle()`. Не забудьте отладить код, рисующий холст, перед тем, как перемешивать карточки, это упростит отладку.

4. Модифицируйте обработчик рисования так, чтобы рисовался пустой зелёный прямоугольник либо значение карточки. Для реализации такого поведения программы предлагается создать второй список с названием `exposed`, каждая запись которого имеет значение `True`, если соответствующая ей карта перевёрнута и её значение видно, и `False`, если карта лежит рубашкой вверх и её значение недоступно. Для проверки кода следует инициализировать список `exposed` какой-либо последовательностью значений.

5. Далее добавьте функциональность, позволяющую определить, на какую из карт нажали мышкой. Для этого добавьте обработчик событий кликов мыши, который получает позицию клика мыши и печатает индекс карточки, которая была кликнута, на консоли. Чтобы упростить определение, какая из карт выбрана, в шаблоне использован такой размер холста, чтобы карты его полностью равномерно заполняли.

6. Измените обработчик событий кликов мышки так, чтобы он переворачивал карточку, которую щелкнули мышкой. Если игрок кликнул `i`-ю карточку, следует изменить значение `exposed[i]` с `False` на `True`. Если карта до клика уже была открыта – игнорируйте этот клик мыши. На данном этапе основа для игры написана.

7. Далее следует добавить логику к обработчику щелчка мыши для выбора двух карточек и определения, совпадают они или нет.
   Совет: Введите глобальную переменную, которая будет отслеживать состояние игры со следующими значениями:

   a) состояние 0 соответствует началу игры,
   b) 1 – открытию первой карты во время хода,
   c) 2 – открытию второй карты во время хода (ход завершен).

   Игра начинается в состоянии 0. После первого щелчка мыши игра переходит в состояние 1. Далее после каждого щелчка по закрытой карте она переходит между состояниями 1 и 2 по цепочке $2 \to 1 \to 2 \to 1 \to 2 \to \cdots$. Такое поведение можно реализовать следующим кодом (не забудьте добавить проверку на попадание в закрытую карту):

```
def click(pos):
    global state
```

```
if state == 0:
    state = 1
elif state == 1:
    state = 2
else:
    state = 1
```

8. Заметим, что в состоянии 2 вам следует определить, не были ли парными две предыдущие карты. Если они были непарными, вам следует их опять перевернуть вверх рубашкой. Для хранения индексов обеих карт, выбранных в предыдущий ход, следует использовать глобальные переменные.

9. Добавьте счётчик, отслеживающий число ходов. Используйте функцию `set_text()` для метки на панели управления для обновления этого счётчика.
   Для создания текстовой метки необходимо после создания окна (`frame`) выполнить
   ```
   label = frame.add_label("Ход = 0")
   ```
   После этого для обновления текста метки используйте
   ```
   label.set_text("Ход = " + str(turn))
   ```

10. В завершении реализуйте функцию `new_game()` так, чтобы кнопка «Перезапустить игру» перемешивала карты, обнуляла счётчик ходов и перезапускала игру. В начале игры все карты должны быть закрыты.

Дополнительные задания

1. Замените `draw_text()` на `draw_image()` для каждой карты, используя 8 различных изображений с цифрами.
2. Организуйте свой код так, чтобы можно было изменять число карт в игре, меняя только значение `NUM_PAIRS` в начале шаблона.

На первый взгляд проект может показаться сложным, но окончательная реализация, проведённая согласно пунктам, занимает около 100 строк кода с комментариями и пробелами. Если у вас возникают трудности, сосредоточьтесь на разработке вашего проекта до 6-го шага, т.к. на этом пункте проще увидеть игру целиком. Далее разработка пойдёт проще.

Оценка задания

В начале игры правильно производится перемешивание карт – 10 %.

Игра рисует карточки с цифрами на холсте – 10 %.

В начале игры щелчок по закрытой карте приводит к её открытию – 10 %.

Игнорируются щелчки на открытых картах – 10 %.

Если открыта одна непарная карта (состояние=1), то щелчок на закрытой карте приводит к её открытию – 10 %.

Если открыты две **непарные** карты (состояние=2), то щелчок на закрытой карте приводит к её открытию и закрытию непарных карт – 10 %.

Если открыты две **парные** карты (состояние=2), то щелчок на закрытой карте приводит к её открытию, парные карты остаются открытыми – 10 %.

Парные карты, открытые на одном ходе, остаются открытыми до перезапуска игры – 10 %.

Игра ведёт корректный подсчёт ходов (номер хода может увеличиваться или после первого, или после второго выбора игрока внутри хода) – 5 %.

Номер хода отображается в текстовой метке на панели управления – 10 %.

Имеется кнопка перезапуска игры, при нажатии на которую счётчик числа ходов сбрасывается, карты перемешиваются и закрываются – 5 %.

Дополнительные баллы

Для вывода карт используются изображения вместо текста – 10 %.

Число карт может быть изменено переменной `NUM_PAIRS` – 10 %.

7. Блэкджек

Постановка задачи

Блэкджек – простая карточная игра. В игру играют двое – игрок и дилер. В игре каждая карта имеет свою стоимость. Задачей игрока является набрать больше очков, чем у дилера.

В блэкджек играют колодой из 52 карт (в каждой масти карты $2, \dots, 10$, валет, дама, король и туз). У каждой карты есть своя стоимость: от 2 до 10 – соответственно $2, \dots, 10$ очков, у картинок (валет, дама и король) – 10 очков, у туза – 1 или 11 очков по желанию игрока.

В начале игры колода перемешивается. Игроку и дилеру изначально раздают по 2 карты, одна карта дилера открыта, а другая – закрыта.

Игрок смотрит свои карты и может попросить ещё карту (англ. hit) или остановиться (англ. stand). Если сумма очков игрока превысит 21, то говорят, что произошёл перебор (англ. bust) – в этом случае игрок сразу проигрывает.

После того как игрок остановился, дилер открывает свою закрытую карту. Если очки дилера меньше 17, он набирает карты, пока его очки не превысят 17. Если сумма очков дилера превысит 21, он проигрывает.

Выигрывает тот, кто набрал больше очков, не превысив при этом 21. В нашей версии при равенстве очков выигрывает дилер.

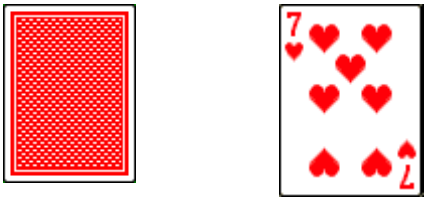
Для упрощения работы рекомендуется использовать шаблон memory_template.py.

Пример игры

Начало игры

Колода перемешана. Игрокам раздают по две карты. Одна карта дилера закрыта.

Карты дилера



Карты игрока

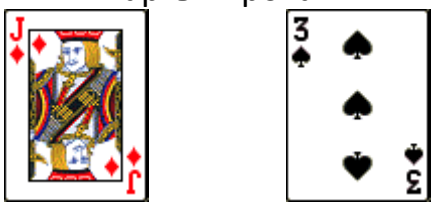
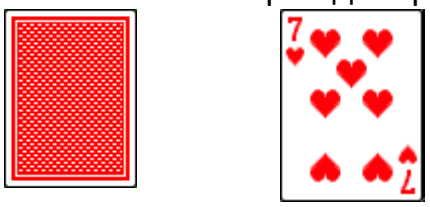


Рис. 7.1. Карты в начале игры

Ходы игрока

Сумма очков игрока – 13 (валет – 10 очков, тройка – 3 очка). Игрок решает взять ещё карту.

Карты дилера



Карты игрока

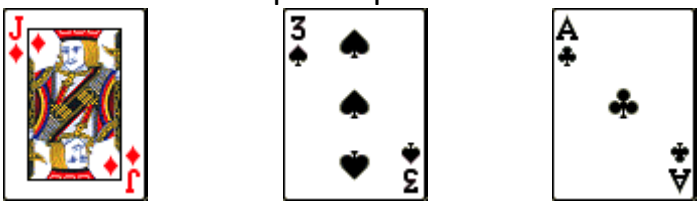
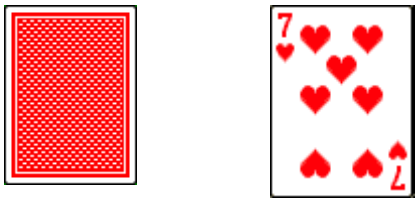


Рис. 7.2. Карты после первого выбора игрока

Игрок может использовать туз только как 1 очко (иначе он получит $10 + 3 + 11 = 24$ очка и проиграет). Сумма очков игрока становится $10 + 3 + 1 = 14$. Игрок решает взять ещё одну карту.

Карты дилера



Карты игрока

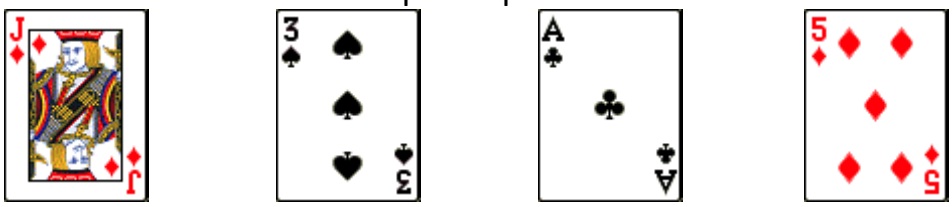


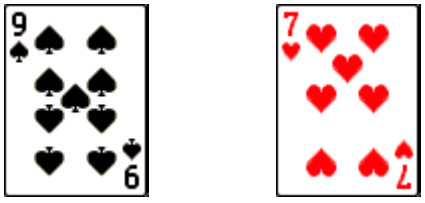
Рис. 7.3. Карты после второго выбора игрока

Сумма очков игрока достигла 19. Игрок решает остановиться.

Ходы дилера

Дилер открывает свою закрытую карту.

Карты дилера



Карты игрока

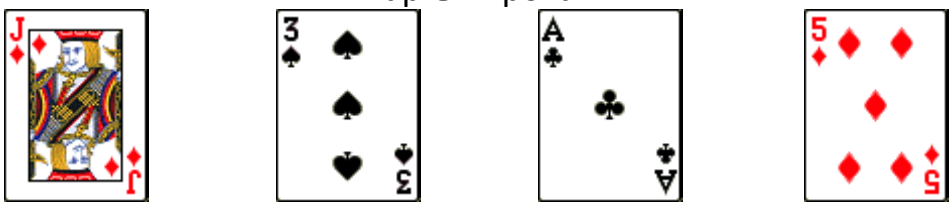


Рис. 7.4. Карты в начале хода дилера

Сумма очков дилера: $7 + 9 = 16$. Так как у дилера меньше 17 очков, он должен взять ещё одну карту.

Карты дилера

Карты игрока

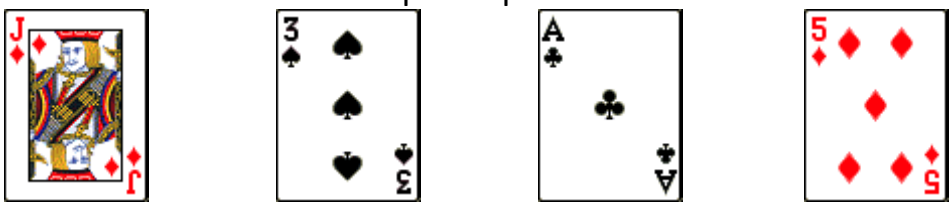


Рис. 7.5. Карты в конце ходов дилера

Сумма очков дилера: $9 + 7 + 4 = 20$. Дилер набрал ≥ 17 очков и должен остановиться.

Определение победителя

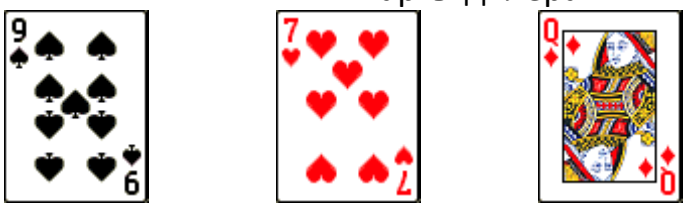
У игрока 19 очков. У дилера 20 очков. У дилера больше, чем у игрока, дилер выигрывает.

Перебор

Допустим, на последнем шаге дилер получит даму (рис. 7.6).

В этом случае сумма очков дилера составит $9 + 7 + 10 = 26$ очков, что больше 21. Дилер проигрывает.

Карты дилера



Карты игрока

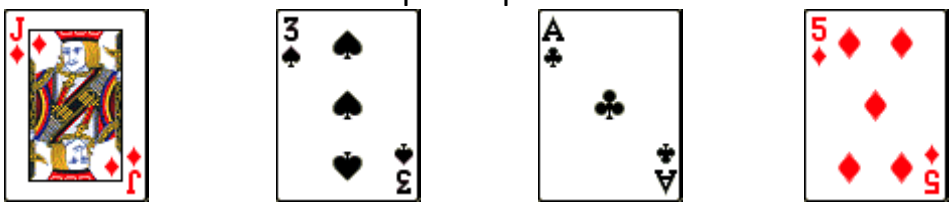


Рис. 7.6. Перебор у дилера

Наблюдения за ходом игры

Обратите внимание на следующие моменты.

- У игрока есть два действия – взять ещё карту (англ. hit) и остановиться (англ. stand).
- При показе игрового поля во время хода игрока закрыта одна карта дилера. После того, как игрок остановится, эта карта открывается.
- Обеим сторонам выгодно использовать стоимость туза как 11, до тех пор, пока полученная в результате сумма не превысит 21. После этого туза можно использовать только как 1. Поэтому стоимость туза зависит не от произвола игрока, а от наличия других карт.
- Второго туза всегда можно использовать только как 1 (сумма очков за два туза: $11 + 11 = 22 > 21$). Как 11 может быть засчитан только один туз.

Классы для игры «Блэкджек»

Рассмотрим основные объекты, манипуляции с которыми происходят в игре блэкджек:

- игральные карты (англ. card);
- колода (англ. deck) – набор карт, откуда их выдают;
- рука (англ. hand) – набор карт, выданных игроку или дилеру.

В предложенном шаблоне для каждого из этих объектов создан класс. В качестве имени классов были использованы термины английского языка.

Для представления одной игральной карты используется класс `Card`. В карточных играх существенной информацией, идентифицирующей карту, являются её масть и достоинство (тройка, семёрка, туз, и т.п.). Соответственно поля данных этого класса включают `suit` – масть и `rank` – достоинство[8]. Для вывода карты на экран могут потребоваться и другие поля, специфичные именно для компьютерной модели карты, такие, как изображение лица (в этом проекте такие поля не будут использоваться).

[8] Для представления мастей в шаблонах игры и тестов используются латинские буквы: `C` – крести (♣), `S` – пики (♠), `H` – черви (♥), `D` – буби (♦). Для обозначения достоинств используются латинские буквы: `A` – туз, `T` – десятка, `J` – валет, `Q` – дама, `K` – король. Остальные достоинства обозначаются соответствующими цифрами (не числами!).

Игральная карта не имеет сложного поведения, которое меняло бы её внутреннее состояние. Поэтому в предложенном варианте у класса Card есть методы для узнавания масти и достоинства, а также метод для рисования карты на холсте.

Содержанием классов Deck и Hand является набор карт. С точки зрения содержания эти классы можно было бы объединить (например, под названием «набор карт»). Однако они обладают существенно различающимся поведением. Поэтому лучше реализовать эти классы отдельно.

Класс Deck (колода) должен обеспечить методы для перемешивания колоды и выдачи очередной карты (при выдаче карта удаляется из колоды). При создании объекта этого класса его конструктор должен заполнить объект полным набором карт.

Класс Hand (рука) содержит методы для добавления карты и определения суммы очков. При создании объекта он должен быть инициализирован пустым списком карт. В классе Hand также требуется метод для рисования.

В таблице 7.1 представлена сводка по классам для игры «Блэкджек», их данным и методам.

Таблица 7.1. Классы для игры «Блэкджек»

Класс	Данные	Методы
Card	`suit` – масть; `rank` – достоинство.	`get_suit()` – узнать масть, `get_rank()` – узнать достоинство, `draw(canvas, pos)` – нарисовать карту на холсте в заданной позиции.
Deck	Набор карт.	`shuffle()` – перемешать, `deal_card()` - выдать карту.
Hand	Набор карт.	`add_card()` – добавить карту, `get_value()` – узнать сумму очков, `draw(canvas, pos)` – нарисовать на холсте в заданной позиции.

При проектировании системы классов стараются разделять задачи между классами таким образом, чтобы максимально обеспечить возможность повторного использования классов в других проектах. Например, класс Card

может быть использован без изменений в любой карточной игре с 52-карточной колодой.

Заметим, что класс `Card` никак не участвует в подсчёте очков – за это полностью отвечает класс `Hand`. Это разумно, так как подсчёт очков – свойство конкретной игры, в других играх стоимость карт может быть другой, или её может вообще не быть. Если бы подсчёт очков был помещён в классе `Card`, то его пришлось бы перерабатывать для использования в других играх.

Кроме того, в игре «Блэкджек» стоимость туза зависит от других карт руки. Это также свидетельствует о том, что помещать расчёт стоимости следует в классе `Hand`.

Класс `Hand` в данном проекте привязан к конкретной игре – он обеспечивает хранение карт и подсчёт очков по правилам блэкджек. С этой точки зрения его правильнее называть «рука для блэкджек», а не просто «рука». Для реализации других карточных игр придётся разрабатывать похожий класс, который может отличаться методами и логикой работы.

Не следует слишком увлекаться «универсализацией» классов. Например, класс `Card` можно сделать более универсальным, добавив возможность использовать расширенные наборы достоинств (например, джокер) или наоборот только сокращенную колоду из 36 карт. Однако вполне может быть, что использовать этот класс для других игр никогда не потребуется. Тогда работа, потраченная на реализацию этих функций, пропадёт зря, а использование такого класса в проекте «Блэкджек» окажется немного сложнее (надо будет выбрать правильную колоду).

Поэтому, определяя функции для классов в проекте, ориентируйтесь на те потребности, которые реально есть в существующем проекте. Чтобы понять, в каком классе должна быть реализована каждая функция, подумайте о других возможных использованиях каждого класса. Для более подробного знакомства с принципами объектно-ориентированного дизайна можно обратиться к работе [10].

Рекомендованный порядок работы над заданием

Рекомендуется разбить работу над реализацией игры на две части. В первой части сосредоточьтесь на реализации логики игры. Для вывода

информации используйте консоль. Графический интерфейс лучше добавить во второй части.

Реализация логики игры

1. Откройте шаблон blackjack_template.py и ознакомьтесь с классом `Card` (карта). Этот класс уже реализован, Ваша задача – изучить его интерфейс и код. Скопируйте реализацию класса `Card` в шаблон card_test.py и проверьте, что все тесты работают корректно.

2. Реализуйте методы `__init__()`, `__str__()`, `add_card()` в классе `Hand` (рука). Рекомендуется использовать список карт для моделирования руки.
   При реализации метода `__str__()` используйте функцию `str()` для получения описания каждой карты (`Card`).
   Проверьте свою реализацию класса `Hand` с помощью шаблона hand_test.py.

3. Реализуйте методы класса `Deck` (колода) согласно имеющемуся шаблону.
   Для моделирования колоды используйте список карт. Начните с создания всех карт с помощью двух вложенных циклов: один по масти (англ. suit), а другой по категории (англ. rank). Для создания каждой карты используйте конструктор класса `Card`.
   Для перемешивания колоды используйте функцию `random.shuffle()`.
   Проверьте реализацию класса Deck, используя шаблон deck_test.py.

4. Реализуйте обработчик для кнопки «Раздать» (англ. deal), который должен перемешать карты и раздать по две карты игроку и дилеру.
   Этот обработчик должен перемешать колоду (её следует хранить как глобальную переменную), создать новые руки для игрока и дилера (их также следует хранить как глобальные переменные), и добавить в каждую руку по две карты.
   Для добавления карты следует использовать метод `deal_card()` класса `Deck` и метод `add_card()` класса `Hand`.
   Полученные руки должны быть распечатаны на консоль с сообщением, показывающим, какая рука принадлежит игроку, а какая – дилеру.

5. Реализуйте метод `get_value()` класса `Hand`. Следует использовать словарь VALUE (имеется в шаблоне) со значениями карт.

Для расчёта стоимости руки используйте следующий метод. Вначале сложите стоимость всех карт, считая стоимость туза за 1. Во время сложения запомните (используя переменную булевского типа), встречался ли в руке туз. Если туз был, добавьте к сумме `10`, если это не приводит к «перебору» (полученная сумма не должна превысить 21).
Чтобы понять, почему этот метод подсчёта очков работает корректно, обратитесь к разделу «Наблюдения за ходом игры».
Проверьте реализацию метода `get_value()`, используя шаблон value_template.py.

6. Реализуйте обработчик для кнопки «Взять ещё» (англ. hit). Если значение руки игрока ≤ 21, обработчик должен добавить в неё ещё одну карту. Если значение руки превысит 21, на консоли должно быть напечатано сообщение «Перебор».

7. Реализуйте обработчик для кнопки «Хватит» (англ. stand). Если игрок перебрал, напомните ему об этом. В противном случае в цикле добавляйте карты в руку дилера, пока её стоимость не станет ≥ 17. Используйте цикл `while`.
Если дилер перебрал, распечатайте соответствующее сообщение. Иначе сравните очки игрока и дилера и распечатайте сообщение о победителе. Помните, что в разрабатываемой версии игры при равенстве очков выигрывает дилер.

В предложенном варианте игра автоматически раздаёт карты игроку и дилеру при запуске. Это реализовано с помощью вызова функции `deal()` во время инициализации.

После реализации этой части программы рекомендуется провести её тщательное и всестороннее тестирование.

Реализация интерфейса

Убедившись, что логика программы реализована правильно, переходите ко второму этапу разработки.

При реализации функций, подразумевающих вывод на экран значений глобальных переменных, не забывайте инициализировать их начальные значения (такие, как создание пустой руки для игрока и дилера) непосредственно перед запуском окна.

1. Реализуйте метод `draw()` класса `Hand`, используя для рисования каждой карты одноимённый метод класса `Card`.
   Рекомендуется рисовать руку как последовательность карт, расположенных в одну строку по горизонтали. Параметр `pos` используйте как координаты верхнего левого угла самой левой карты (рис. 7.7).
   Для упрощения кода считайте, что на холсте должно быть видно не более 5 карт игрока.

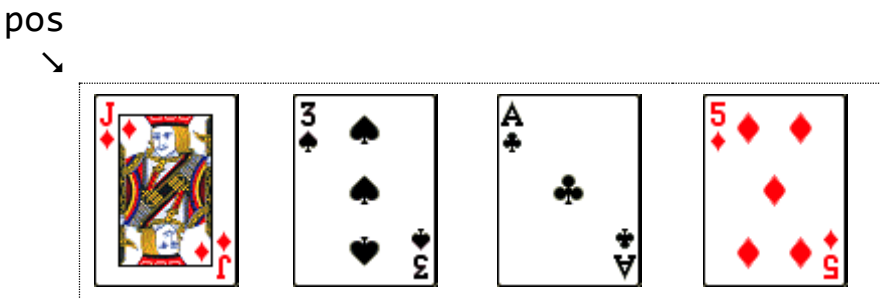


Рис. 7.7. Вывод руки

2. Замените вывод сообщений в консоли отображением сообщения на холсте. Рекомендуется использовать глобальную строковую переменную `outcome` (от англ. исход), которую следует выводить в обработчике рисования `draw()` с помощи функции `draw_text()`. Сообщения должны запрашивать действия у пользователя и иметь форму вопроса, например: «Ещё или хватит?», «Раздать снова?».
   Нарисуйте название игры «Блекджек» на холсте.

3. Добавьте глобальную булеву переменную `in_play`, которая должна отслеживать, продолжается ли ещё ход игрока. Если игрок все ещё продолжает выбирать, следует рисовать рубашку перевёрнутой карты поверх первой карты руки дилера. Когда игрок останавливается, следует прекратить скрывать руку дилера.

4. Добавьте счётчики числа выигрышей и поражений для сессии игры.

5. Измените логику обработчика кнопки «Раздать», чтобы он каждый раз создавал и перемешивал новую колоду. Это позволит избежать ситуации, когда колода становится пустой во время игры.

6. Измените логику обработчика копки «Раздать», чтобы при нажатии на эту кнопку во время игры, программа сообщала о проигрыше игрока и соответственно корректировала очки.

Дополнительные задания

Реализуйте последовательный вывод карт руки дилера, после того, как игрок остановится. Выводите каждую новую карту дилера через секунду. Используйте таймер и глобальную переменную, отслеживающую, сколько карт можно показывать. Максимальное число карт для отображения можно передавать в качестве параметра функции `draw()` класса `Hand`.

Оценка задания

Игра отображает руки игрока и дилера текстовыми сообщениями – 10 %

или

игра отображает руки игрока и дилера в графическом режиме – 20 %.

Первая карта дилера скрыта во время хода игрока – 10 %.

Нажатие на кнопку «Раздать» перемешивает колоду и раздаёт по две карты – 20 %.

Нажатие на кнопку «Раздать» в процессе игры засчитывается как проигрыш – 5 %.

Нажатие на кнопку «Ещё» приводит к выдаче карты игроку – 10 %.

Нажатие на кнопку «Хватит» приводит к выдаче карт дилеру по правилам игры – 10 %.

Программа распознаёт перебор игрока и дилера – 10 %.

Программа корректно определяет и объявляет победителя – 5 %.

Программа корректно выводит приглашения вида «Ещё или хватит?», «Раздать снова?» – 5 %.

Программа корректно ведёт счёт выигрышей и проигрышей – 5 %.

Дополнительные баллы

Последовательный вывод карт дилера в конце раунда – 15 %.

8. Космический корабль

Постановка задачи

Вам предстоит разработать аналог игры «Астероиды». В разрабатываемой версии игрок управляет космическим кораблём с помощью 4-х кнопок: 2 кнопки для поворота (по часовой и против часовой стрелке), третья кнопка для придания кораблю скорости, и ещё одна позволяет стрелять ракетами в препятствия. По экрану летают большие астероиды в случайном направлении со случайной скоростью. Цель игры состоит в том, чтобы разрушить все астероиды до столкновения с космическим кораблём.

В оригинальной аркадной версии большие астероиды при попадании в них ракетой распадались на мелкие части, которые также могут повредить корабль и которые следует уничтожать. Периодически появлялась летающая тарелка, которая стремилась уничтожить летающий корабль.

Разработка игры будет разбита на два домашних задания. В первом необходимо разработать прототип игры, включающий летающий космический корабль, один астероид и одну ракету. Шаблон программы содержит необходимый код и графику для того, чтобы игра выглядела современнее.

Рекомендованный порядок работы над заданием

Процесс создания прототипа игры следует разделить на четыре этапа.

Этап 1. Космический корабль

На этом этапе необходимо реализовать схему управления космическим кораблём, для чего потребуется работать с классом `Ship` (космический корабль) и обработчиком нажатия на клавиатуру.

Корабль должен реагировать на нажатие кнопок следующим образом.

- Стрелки влево и вправо контролируют ориентацию корабля в пространстве. При нажатии стрелки влево корабль должен поворачиваться против часовой стрелки. При нажатии стрелки вправо – по часовой. Если не нажата ни одна из этих кнопок, корабль не должен менять свою ориентацию в пространстве. Вам потребуется подобрать разумную угловую скорость для поворота корабля.

- Стрелка вверх контролирует работу двигателя космического корабля. Двигатель включается при нажатии кнопки вверх и выключается, если кнопка отпущена. Во время включённого двигателя следует рисовать огонь у турбин корабля. Если двигатель не включен, то огня быть не должно.

- Когда двигатель включён, корабль должен ускоряться в направлении «прямо по курсу». Вектор направления может быть рассчитан в зависимости от угла корабля с помощью функции `angle_to_vector()`. Следует поэкспериментировать с масштабированием компонент вектора ускорения для определения разумного ускорения корабля.

- Следует помнить, что во время движения корабль ускоряется в прямом направлении, но при этом двигается в зависимости от вектора скорости. Эти два направления могут не совпадать. Возможность ускориться в направлении, отличном от текущего направления движения, является отличительной чертой игры «Астероиды».

- Необходимо учитывать силу трения при движении корабля. Для этого при обновлении скорость всегда нужно умножать на константу, меньшую 1, что будет тормозить корабль. После остановки двигателя корабль через некоторое время должен остановиться.

Рекомендуется организовать работу над реализацией управления кораблём в следующем порядке.

1. Отредактируйте метод рисования для класса `Ship` для вывода изображения корабля (без огня у турбин) вместо рисования кружочка. Метод должен учитывать позицию и угол корабля. Помните, что угол должен считаться в радианах, а не в градусах. Если метод рисования корабля вызывается из обработчика рисования, то корабль должен отображаться на экране.
   Поэкспериментируйте с различными позициями и углами корабля, чтобы убедиться, что рисование работает корректно.

2. Разработайте первоначальную версию метода обновления позиции корабля. Она должна обновлять позицию корабля, основываясь на его скорости.
   Если метод обновления вызывается из обработчика рисования, корабль

должен начать двигаться. Проверьте движение корабля с различными начальными скоростями.

3. Модифицируйте метод обновления для корабля так, чтобы изменять его угол ориентации в соответствии с текущим значением угловой скорости.

4. Научите корабль двигаться в зависимости от нажатия на правую/левую стрелки.
   Для этого добавьте методы в класс `Ship` для увеличения и уменьшения угловой скорости на фиксированную величину (подберите по вашему усмотрению).
   Добавьте обработчик нажатия и отпускания кнопок, который должен реагировать на кнопки вверх/вниз и вызывать соответствующие методы класса `Ship`.
   Проверьте, что корабль двигается, как задумано.

5. Модифицируйте обработчик клавиатуры для включения/выключения огня у турбин корабля. Добавьте метод к классу `Ship`, отвечающий за включение и выключение двигателя. Состояние двигателя можно передать с помощью аргумента булевского типа.

6. Модифицируйте метод рисования корабля для рисования огня, когда двигатель включен.

7. Модифицируйте метод включения двигателя корабля для проигрывания звука работающих турбин, когда двигатель включен. При выключении двигателя перематывайте звук в начало.

8. Добавьте код к методу обновления позиции корабля для учёта ускорения при включённом двигателе.
   Используйте вспомогательную функцию `angle_to_vector()` для получения вектора ориентации корабля по углу поворота. Этот вектор является вектором ускорения корабля.
   Обновите вектор скорости с учётом вектора ускорения. Вам потребуется умножить вектор ускорения на небольшое число, чтобы корабль не слишком сильно ускорялся.

9. Далее модифицируйте метод обновления корабля таким образом, чтобы при пересечении им края экрана корабль появлялся на противоположном крае (используйте вычисления по модулю).

10. На данный момент корабль не замедляется. Добавьте трение к методу обновления корабля. Для этого при каждом обновлении умножайте каждую из компонент скорости на число, меньшее 1.

Итак, реализован корабль, который умеет летать по экрану. Подберите значения констант так, чтобы управление было удобным для игрока.

Этап 2. Астероид

Для создания астероидов будет использован класс `Sprite`. Обратите внимание, что метод обновления позиции для спрайта очень напоминает метод обновления для корабля. Главное отличие состоит в том, что скорость корабля и поворот контролируются с помощью клавиш, в то время как спрайты движутся в соответствии с параметрами, установленными случайно при их создании. Выходя за границы экрана, астероиды должны появляться с противоположной его стороны, так же как и корабль.

В шаблоне при старте игры создаётся один астероид с нулевой скоростью (переменная `a_rock`). Вместо этого следует создавать астероид, заменяя `a_rock` каждую секунду в обработчике таймера.

В следующем задании игра будет работать с несколькими астероидами. В этой версии астероиды не таранят корабль, соприкасаясь с ним.

Работу над астероидом выполняйте в следующем порядке.

1. Завершите реализацию класса `Sprite`.
   Модифицируйте обработчик рисования, чтобы он рисовал изображение астероида.
   Реализуйте метод обновления, чтобы спрайт двигался и крутился. Астероиды не ускоряются и не имеют трения, так что этот метод обновления должен быть проще, чем у корабля.
   Проверьте правильность работы, подавая на вход разные начальные параметры, и убедитесь, что всё работает правильно.

2. Модифицируйте обработчик таймера `rock_spawner()`. Устанавливайте в качестве значения `a_rock` новый астероид каждый такт. Не забудьте

объявить `a_rock` как глобальную переменную в обработчике таймера. При создании астероида случайно генерируйте скорость, положение и угловую скорость. Вам потребуется подобрать рамки для случайных величин, чтобы в игру было интересно играть. Удостоверьтесь, что астероиды крутятся и двигаются во всех направлениях.

Этап 3. Ракета

Ракеты также будут реализованы на основе класса `Sprite`, как и астероиды. Ракеты всегда имеют нулевую угловую скорость. У них есть время жизни (они должны исчезнуть по прошествии времени, иначе ракеты будут повсюду), но пока эта функция не будет реализована. На данный момент необходимо сделать так, чтобы можно было запустить 1 ракету, которая не разрушает астероиды. Взрывы будут добавлены позже.

Ракета должна вылетать по нажатию на пробел, а не по таймеру, как астероиды. Она также должна, вылетая за границы экрана, появляться с другой стороны.

Реализуйте ракету в следующем порядке.

1. Добавьте метод `shoot()` (англ. выстрел) к классу «корабль». Этот метод должен выпускать новую ракету (на данный момент необходимо поменять существующую ракету в переменной `a_missile`). Начальная позиция ракеты должна совпадать с координатой кончика пушки корабля. Её скорость является суммой скорости корабля и нескольких векторов направления.
2. Модифицируйте обработчик нажатия на кнопку для вызова метода `shoot()`, если нажата кнопка пробела.
3. Убедитесь, что звук ракет воспроизводится во время выстрела.

Этап 4. Интерфейс игры

Интерфейс игры показывает количество оставшихся жизней и очки (результат). Их подсчёт отложите до следующего задания. Добавьте код к обработчику рисования для рисования показателей на холсте. Используйте глобальные переменные `lives` и `score` для хранения оставшихся жизней и очков.

Дополнительные задания

Добавьте второй слой анимированного фона, который будет находиться под существующим и двигаться медленнее. Возможно, стоит разделить астероиды в этих слоях по размерам: в более быстром (и близком) слое – покрупнее, в более медленном (дальнем) – помельче.

Оценка задания

Программа отображает изображение корабля	– 5 %.
Корабль летит по прямой, если двигатель выключен	– 5 %.
Корабль вращается при нажатии на клавиши ← и →	– 10 %.
Ориентация корабля не зависит от его скорости	– 5 %.
При нажатии клавиши ↑ программа рисует корабль с огнём двигателей	– 5 %.
Программа воспроизводит звук двигателя при нажатой клавише ↑	– 5 %.
Корабль ускоряется в направлении своей ориентации при нажатой клавише ↑	– 5 %.
При пересечении границы экрана корабль появляется с другой стороны	– 5 %.
Корабль останавливается, если двигатели выключены	– 5 %.
Программа отображает изображение астероида	– 5 %.
Астероид движется прямолинейно с постоянной скоростью	– 5 %.
Астероид пересоздаётся каждую секунду	– 5 %.
Астероид создаётся со случайными позицией, скоростью и угловой скоростью	– 5 %.
При нажатии на пробел выпускается ракета	– 5 %.
Ракета создаётся на конце пушки корабля	– 5 %.
Скорость ракеты определяется на основе скорости и направления корабля	– 5 %.

При запуске ракеты программа проигрывает соответствующий звук – 5 %.

Программа выводит число жизней в левом верхнем углу экрана – 5 %.

Программа выводит очки в правом верхнем углу экрана – 5 %.

Дополнительные баллы

Второй слой анимированного фона – 15 %.

9. Астероиды

Постановка задачи

В этом домашнем задании будет продолжена работа над игрой «Астероиды».

В ходе решения задания игра будет дополнена возможностями работы с несколькими астероидами и ракетами одновременно. Игрок будет терять «жизни» при столкновении корабля с астероидом и набирать очки, сбивая астероиды ракетами. Потребуется вести подсчёт жизней и завершать игру, когда они закончатся. В качестве дополнительного задания можно будет реализовать анимацию взрывов.

Код шаблона для этого домашнего задания содержит некоторые дополнения по сравнению с предыдущим шаблоном. Скопируйте в него реализацию классов `Ship` и `Sprite`, а также обработчики клавиатуры и таймера из предыдущего задания.

Рекомендованный порядок работы над заданием

Процесс доработки игры можно разделить на пять этапов.

Этап 1. Несколько астероидов

Вначале реализуйте поддержку работы с несколькими астероидами.

Для этого действуйте в следующем порядке:

1. Удалите переменную `a_rock`, а вместо неё введите переменную для множества астероидов `rock_group`.
   Инициализируйте `rock_group` пустым множеством.
   Модифицируйте код создания астероидов (обработчик таймера) так, чтобы он каждый раз создавал новые астероиды (объекты класса `Sprite`) и добавлял их в `rock_group`.

2. Модифицируйте код создания астероидов так, чтобы он не создавал слишком много астероидов. Рекомендуется ограничить число одновременно используемых астероидов двенадцатью. При большем числе астероидов игра становится менее интересной.
   Для этого проверяйте, сколько астероидов уже есть в `rock_group`, и создавайте новый, только если лимит ещё не набран.

3. Создайте вспомогательную функцию `process_sprite_group()`. Аргументами этой функции должны быть множество спрайтов и холст. Для каждого спрайта функция должна вызвать методы `update()` и `paint()`.

4. Вызовите функцию `process_sprite_group()` из обработчика рисования. Эта функция должна обрабатывать множество астероидов `rock_group`.

В конце первого этапа в игре каждую секунду должен создаваться новый астероид. При достижении заданного лимита появление новых астероидов должно прекратиться. Все астероиды должны перемещаться независимо друг от друга.

Этап 2. Столкновения

Теперь добавьте столкновения между кораблём и астероидами.

1. Добавьте метод `collide()` (англ. столкновение) к классу `Sprite`.
   Аргументом метода должен быть `other_object` – «другой объект», столкновение с которым проверяется.
   Метод должен возвращать значение `True`, если зафиксировано столкновение, и `False` – если нет.
   Пока в качестве «другого объекта» будет выступать корабль, но далее эта же функция будет использована для проверки столкновений с ракетами.
   Для определения столкновения используйте координаты центров и радиусы объектов. Для получения этих данных добавьте в классы `Ship` и `Sprite` пары методов `get_position()` и `get_radius()`.

2. Реализуйте вспомогательную функцию `group_collide()`.
   Аргументами этой функции должны быть множество спрайтов group и спрайт `other_object`. Функция должна для всех спрайтов группы проверить столкновение с `other_object`.
   Для проверки столкновения используйте метод `collide()` класса `Sprite`.
   Если зафиксировано столкновение, то столкнувшийся объект должен быть удалён из группы. Не забудьте, что внутри цикла `for … in …` нельзя модифицировать структуру, по которой проводится итерация. Используйте один из вариантов удаления, рассмотренный в примере balls-del.py.

Функция должна возвращать значение `True`, если зафиксировано хотя бы одно столкновение, и `False` – если нет.

3. В обработчике рисования используйте функцию `group_collide()` для проверки столкновения корабля с астероидами. В случае столкновения уменьшайте число жизней на 1 (переменная `lives`). Пока допустима игра с отрицательным числом жизней.

   На этом этапе игрок может спасаться от астероидов, только улетая от них.

Этап 3. Ракеты

Добавьте поддержку работы с группой ракет.

1. Удалите переменную `a_missile` и введите вместо неё переменную `missile_group` для множества ракет.
   Инициализируйте `missile_group` пустым множеством.
   Модифицируйте метод выстрела `Shoot()` в классе `Ship` так, чтобы при каждом выстреле создавалась новая ракета (объект класса `Sprite`) и добавлялась к множеству `missile_set`.

2. В обработчик рисования добавьте вызов функции `process_sprite_group()` для обработки `missile_group`.
   Теперь при каждом выстреле игрока возникает новая ракета. Но ракеты остаются на всё время игры. Срок их жизни надо ограничить. Для этого потребуется сделать модификации в классе `Sprite` и в функции `process_sprite_group()`.

3. В методе `update()` класса Sprite увеличивайте на 1 возраст спрайта (переменная `self.age`).
   Если возраст станет больше, чем допустимое время жизни (переменная `self.lifespan`), то такой объект следует удалить. Чтобы это можно было сделать, метод `update()` должен возвращать значение `False` (что будет означать «удалять пока не надо»), если возраст меньше предельного, и значение `True` (объект требует удаления), если достигнуто предельное время жизни.

4. Добавьте проверку значения, возвращаемого методом `update()` в функцию `process_sprite_group()`. Если метод `update()` вернул значение `True`, удалите спрайт из группы.

Не забудьте о запрете удаления элементов из множества, по которому проходит итерация.

По завершении третьего этапа игрок получил возможность стрелять, однако ракеты пока не сбивают астероиды.

Этап 4. Доработка столкновений

Теперь нужно обеспечить проверку столкновений между ракетами и астероидами. Использовать непосредственно функцию `group_collide()` не получится – необходимо проверять столкновения между двумя группами. Для этого лучше разработать ещё одну вспомогательную функцию.

1. Реализуйте вспомогательную функцию `group_group_collide()`, аргументами которой являются два множества спрайтов.
   В функции реализуйте цикл по копии первого множества. Для каждого спрайта вызывайте `group_collide()` для проверки столкновения со всеми спрайтами второго множества.
   Удаляйте спрайты, для которых зафиксировано столкновение, из первой группы. Для удаления удобно использовать метод `discard()` множества.
   Функция должна вернуть число элементов из первой группы, которые столкнулись с элементами из второй группы.

2. Для проверки столкновений между ракетами и астероидами добавьте вызов функции `group_group_collide()` в обработчик рисования.
   Увеличивайте очки (переменная `score`) за каждый сбитый астероид.

   На этом этапе в игру уже можно играть, но пока нельзя проиграть.

Этап 5. Финальная доработка

Добавьте последние штрихи, чтобы завершить игру.

1. Добавьте код в обработчик рисования, чтобы игра перезапускалась, когда число жизней дойдёт до нуля. Для этого установите значение переменной `started` в `False` и прекратите создание новых астероидов и управление кораблём, пока игра остановлена.

2. Убедитесь, что при перезапуске игры счётчики жизней и очков корректно инициализируются.
   При запуске игры начинайте процесс создания астероидов.

Начинайте воспроизведение музыки в начале игры и перематывайте её при перезапуске.

3. При создании новых астероидов необходимо проверять, чтобы они не создавались слишком близко к кораблю. Иначе корабль может погибнуть от столкновения с возникшим прямо на нём астероидом, что не честно по отношению к игроку.
   Самый простой способ решить эту проблему – игнорировать создание астероида, если он находится слишком близко к кораблю (не добавляйте его в `rock_group`).

4. Попробуйте изменять начальную скорость астероидов пропорционально текущему счёту, чтобы игра усложнялась по мере набора очков.

5. Поэкспериментируйте со значениями разных констант, чтобы сделать игру интересной.

Дополнительные задания

В качестве расширения игры реализуйте анимацию взрывов.

Сохраните копию своего кода перед тем, как попробовать реализовать это расширение.

1. В методе `draw()` класса `Sprite` проверьте, имеет ли переменная `self.Animated` значение `True`.
   Для анимированных спрайтов выберите соответствующий тайл, используя возраст (переменная `self.age`) в качестве индекса, и выводите этот тайл.
   Если спрайт не является анимированным, рисуйте его как прежде.

2. Создайте множество `exploision_group`, инициализируйте его пустым множеством.

3. При обнаружении столкновения в методе `group_collide` создайте новый взрыв (объект класса `Sprite`) и помещайте его в `exploision_group`.
   При каждом взрыве воспроизводите звук взрыва.

4. В обработчике рисования добавьте вызов функции `process_sprite_group()` для обработки взрывов.

Оценка задания

Игра создаёт и рисует несколько астероидов	– 10 %.
Игра проверяет столкновения корабля и астероидов	– 10 %.
После столкновения с кораблём астероид исчезает	– 10 %.
После столкновения корабля с астероидом игрок теряет одну жизнь	– 5 %.
Игра работает с несколькими ракетами	– 10 %.
Ракеты, которые не попали в астероид, исчезают через некоторое время	– 10 %.
Игра проверяет столкновения между астероидами и ракетами	– 10 %.
После столкновения ракеты и астероида оба пропадают	– 10 %.
Игрок получает очки за сбитые астероиды	– 10 %.
Игра завершается и появляется заставка, если число жизней достигает 0	– 5 %.
При щелчке на заставке число жизней устанавливается в 3, очки – в 0, музыка перезапускается	– 5 %.
В режиме заставки игра не создаёт новых астероидов и не управляет кораблём	– 5 %.

Дополнительные баллы

Анимация взрывов	– 10 %.
Звук взрывов	– 5 %.

10. Печеньки

Постановка задачи

Cookie Clicker (Печеньки) – это игра на языке JavaScript [11], разработанная в 2013 году. Это так называемая «игра на увеличение». Смысл игры – испечь как можно больше печенек. Исходно игрок печёт печеньки, кликая мышью на большую печеньку, пока не наберется достаточно количество печенек для покупки строений или улучшений. Купленные строения позволяют получать больше печенек для покупки новых строений или улучшений. И так далее.

В игре есть несколько типов строений. Каждый тип строения имеет собственное значение скорости производства (измеряется в печ/с). Стоимость покупки каждого последующего строения одного типа увеличивается на 15%. Вы можете ознакомиться с оригинальной игрой на сайте http://orteil.dashnet.org/cookieclicker/.

Целью этого задания является разработка простого симулятора игры Cookie Clicker. Вам необходимо будет реализовать различные стратегии и посмотреть, как они будут вести себя на протяжении игры. В симуляторе не будет графического интерфейса, поэтому в нём не будет возможности «кликать». Вместо этого игра начнётся со скорости производства 1 печ/с, а далее можно будет покупать строения по мере накопления печенек. Потребуется реализовать и симулятор, и стратегии для выбора, какие методы производства следует покупать.

В таблице 10.1 представлены строения, которые будут использованы в симуляторе.

Таблица 10.1 Строения в симуляторе Cookie Clicker

Имя строения	Начальная стоимость	Скорость
Курсор	15.0	0.1
Бабушка	100.0	0.5
Ферма	500.0	4.0
Фабрика	3000.0	10.0
Шахта	10000.0	40.0
Космический корабль	40000.0	100.0
Алхимическая лаборатория	200000.0	400.0

Портал	1666666.0	6666.0
Машина времени	123456789.0	98765.0
Конденсатор антиматерии	3999999999.0	999999.0

Предоставляемый код

Для разработки кода следует использовать шаблон clicker_template.py, который содержит описание структуры программы, включая класс `ClickerState`, который должен хранить состояние игры, и функцию `simulate_clicker()`, которая запускает симулятор. В ходе разработки программы может потребоваться добавить дополнительные функции, методы или другой код.

Шаблон включает функцию `run()`, которая запускает симулятор. В исходном виде эта функция просто вызывает функцию `run_strategy()`, которая выполняет моделирование с заданной стратегией. По мере реализации стратегий следует добавлять новые вызовы функции `run_strategy()` в функцию `run()`.

Функция `run_strategy()` выполняет моделирование, печатает состояние игры через заданное число секунд и выводит график производства печенек. Графики могут помочь понять, как число печенек растёт во времени.

Шаблон также содержит реализацию простой стратегии, реализованной в функции `strategy_cursor()`. Обратите внимание, что аргументами этой функции являются:

- `cookies` – текущее число печенек у игрока;
- `cps` – текущая скорость производства печенек;
- `time_left` – время до конца моделирования;
- `build_info` – информация о стоимости строений (см. ниже описание класса `BuildInfo`).

Все функции-стратегии будут получать такие же аргументы.

Стратегия `strategy_cursor()` всегда выбирает покупку строения «Курсор» без анализа текущего состояния. Очевидно, эта стратегия не является эффективной и предоставлена только для отладки остального кода. Обратите внимание, что эта стратегия не проверяет, достаточно ли средств на покупку «Курсора». Стратегии, которые Вам следует реализовать, должны проверять

достаточность средств и возвращать значение None, если никакая покупка не возможна (или нецелесообразна с точки зрения стратегии).

Другая «отладочная» стратегия, которая включена в шаблон – strategy_none(), отказывается от любых покупок.

Вместе с шаблоном предоставляется файл clicker_provided.py, содержащий готовый класс BuildInfo. Этот класс хранит текущие значения скорости производства печенек и стоимости строений, которые может покупать игрок.

При создании объекта класса BuildInfo он инициализируется значениями по умолчанию из игры. После покупки строения его цена в классе BuildInfo будет изменяться согласно правилам игры.

Помните, что если передавать объект класса BuildInfo с помощью оператора = или как параметр функции, то любое изменение в этом объекте будет видно во всей программе. Если такое поведение не подходит, то можно использовать метод clone() класса BuildInfo для получения его независимой копии.

Класс BuildInfo содержит следующие методы:

- build_items() – возвращает список доступных строений (список текстовых строк);
- get_cost(item) – возвращает текущую стоимость строения item;
- get_cps(item) – возвращает скорость производства печенек строением item;
- update_item(item) – обновляет стоимость строения item, этот метод следует вызывать после каждой покупки строения;
- clone() – возвращает копию объекта класса BuildInfo.

Файл clicker_provided.py подключается из файла clicker_template.py с помощью команды import clicker_provided as provided. Для того чтобы эта команда успешно сработала, оба файла должны находиться в одной директории. Внутри файла clicker_template.py объекты класса BuildInfo создаются конструктором provided.BuildInfo(). После создания объекта с ним можно работать так же, как обычно осуществляется работа с объектами:

```
import clicker_provided as provided  # Импортируем файл.
```

```
            …
build_info = provided.BuildInfo() # Создаём объект. Надо писать provided.
            …

c = build_info.get_cost("Бабушка")    # Используем как обычный объект.
```

Установка дополнительного программного обеспечения

В ходе решения этого задания потребуется выводить графики с помощью библиотеки simpleplot. Для её работы требуется установить несколько других библиотек. Следуйте приведённым ниже инструкциям для их установки.

Установка библиотеки Numpy

Библиотека Numpy входит в PyPI, однако её установка включает компиляцию части модулей, написанных на языке C++, что требует наличия компилятора, дополнительных библиотек и соответствующих настроек, занимает много времени и может закончиться неудачно. Поэтому лучше воспользоваться установкой уже скомпилированной библиотеки.

Для установки библиотеки Numpy выполните следующие шаги:

1) скачайте со страницы http://www.lfd.uci.edu/~gohlke/pythonlibs/ файл с последней версией библиотеки (1.9.1 на момент написания инструкции):
   a. numpy-MKL-1.9.1.win-amd64-py3.4.exe если у Вас 64-битный Windows;
   b. numpy-MKL-1.9.1.win32-py3.4.exe если у Вас 32-битный Windows;
2) запустите скаченный файл;
3) нажмите «Далее>»;
4) убедитесь, что установщик правильно нашел путь к Python;
5) нажимайте «Далее>» до завершения установки.

Версия Numpy должна соответствовать установленной версии Python (файлы, оканчивающиеся на …py3.4 предназначены для Python 3.4). Если Вы хотите использовать другую версию Python, скачивайте подходящую версию библиотеки.

Установка библиотеки Matplotlib

Библиотека Matplotlib не входит в PyPI, её нужно скачать с сайта разработчиков и установить.

Для установки библиотеки Matplotlib выполните следующие шаги:

1) скачайте со страницы http://matplotlib.org/downloads.html файл с последней версией библиотеки (1.4.2 на момент написания инструкции):
   a. matplotlib-1.4.2.win-amd64-py3.4.exe если у Вас 64-битный Windows;
   b. matplotlib-1.4.2.win32-py3.4.exe если у Вас 32-битный Windows;
2) запустите скаченный файл;
3) нажмите «Далее>»;
4) убедитесь, что установщик правильно нашел путь к Python;
5) нажимайте «Далее>» до завершения установки.

Версия Matplotlib должна соответствовать установленной версии Python (файлы, оканчивающиеся на …py3.4 предназначены для Python 3.4). Если Вы хотите использовать другую версию Python, скачивайте подходящую версию библиотеки.

Установка библиотеки Six

Эта библиотека входит в Python Package Index (PyPI) и может быть установлена с помощью менеджера пакетов pip. Для установки достаточно выполнить команду `pip install Six`.

Установка должна выглядеть следующим образом:

```
c:\Python34\Scripts>pip install six
Downloading/unpacking six
  Downloading six-1.8.0-py2.py3-none-any.whl
Installing collected packages: six
Successfully installed six
Cleaning up...
```

Установка библиотеки python-dateutil

Эта библиотека входит в Python Package Index (PyPI) и может быть установлена с помощью менеджера пакетов pip. Для установки достаточно выполнить команду `pip install python-dateutil`.

Установка должна выглядеть следующим образом:

```
c:\Python34\Scripts>pip install python-dateutil
Downloading/unpacking python-dateutil
  Running setup.py
(path:C:\Users\sergey\AppData\Local\Temp\pip_build_sergey\python-
dateutil\setup.py) egg_info for package python-dateutil
```

```
Requirement already satisfied (use --upgrade to upgrade): six in
c:\python34\lib\site-packages (from python-dateutil)
Installing collected packages: python-dateutil
  Running setup.py install for python-dateutil

Successfully installed python-dateutil
Cleaning up...
```

Установка библиотеки pyparsing

Эта библиотека входит в Python Package Index (PyPI) и может быть установлена с помощью менеджера пакетов pip. Для установки достаточно выполнить команду `pip install pyparsing`.

Установка должна выглядеть следующим образом:

```
c:\Python34\Scripts>pip install pyparsing
Downloading/unpacking pyparsing
  Running setup.py
(path:C:\Users\sergey\AppData\Local\Temp\pip_build_sergey\pyp
arsing\setup.py) egg_info for package pyparsing

Installing collected packages: pyparsing
  Running setup.py install for pyparsing

Successfully installed pyparsing
Cleaning up...
```

Рекомендованный порядок выполнения задания

Этап 1. Состояние игры

Работу следует начать с реализации класса `ClickerState`. Этот класс инкапсулирует информацию о состоянии игры во время моделирования. Объединение всей информации о состоянии в одном объекте существенно упрощает реализацию симулятора. Класс ClickerState должен хранить информацию о следующих параметрах:

1) общее число печенек, произведённых за всю игру (начальное значение 0.0);
2) число печенек у игрока в данный момент времени (начальное значение 0.0);
3) текущее время игры (начальное значение 0.0);
4) текущая скорость производства печенек в печ/с (начальное значение 1).

Обратите внимание, что все параметры следует хранить в числах типа `float`, так как во время работы симулятора будут получаться дробные значения как скорости, так и числа печенек.

Наряду с этими параметрами класс `ClickerState` должен также хранить историю игры. История должна храниться как список (тип `list`) кортежей (тип `tuple`). Каждый кортеж будет содержать 4 значения:

- время;
- купленное в этот момент времени строение (или `None`, если ничего не покупалось);
- стоимость строения;
- общее число произведённых к этому моменту печенек.

История должна быть инициализирована значением `[(0.0, None, 0.0, 0.0)]`.

Методы класса `ClickerState` должны воздействовать на состояние игры следующим образом:

- `__str__()` – должен возвращать описание состояния в виде, понятном человеку (будет в основном использоваться для отладки);
- `get_cookies()`, `get_cps()`, `get_time()`, `get_history()` – эти методы просто должны возвращать соответственно число печенек, скорость их производства, текущее время и историю;
- `time_until()` – возвращает время, которое должно пройти до накопления заданного числа печенек (обратите внимание, что эта функция всегда должна возвращать целое число секунд);
- `wait()` – этот метод должен «подождать» заданное время, что значит, что в этом методе следует соответственно увеличить текущее время, число печенек у игрока и общее число печенек.
- `buy_item()` – этот метод должен «купить» строение. Это значит, что следует скорректировать число печенек у игрока, скорость производства и добавить соответствующую запись в историю.

При передаче неправильных параметров (например, при попытке купить строение, на которое у игрока недостаточно печенек) методы должны просто вернуть управление, ничего не делая.

1. Выберите имена для хранения параметров, описанных выше.
   Инициализируйте соответствующие переменные класса начальными значениями в методе `__init__()`.

2. Реализуйте метод `__str__()`, с помощью которого можно проверять корректность изменения объекта.
   Метод должен выдавать строку, включающую все параметры состояния игры.
   Убедитесь, что в начальном состоянии описание игры соответствует указанным выше начальным значениям.

3. Реализуйте методы `get_...()`, каждый из которых должен вернуть свою переменную.

4. Реализуйте метод `wait()`. Он должен скорректировать текущее время, и добавить число произведённых за время ожидания печенек к числу печенек у игрока и общему числу печенек.
   Проверьте, что метод корректно пересчитывает параметры.

5. Реализуйте метод `buy_item()`. Он должен уменьшить число печенек у игрока на стоимость покупки, увеличить скорость производства печенек и добавить запись в список истории.
   Не забудьте проверить корректность покупки. Если у игрока недостаточно средств, метод не должен делать никаких действий.

Этап 2. Симулятор

Далее следует реализовать функцию `simulate_clicker()`, которая должна выполнять моделирование игры с заданной стратегией. Эта функция должна получить объект класса `BuildInfo`, время моделирования в секундах и функцию стратегии. Обратите внимание, что `simulate_clicker()` – функция высшего порядка.

1. Скопируйте объект `BuildInfo` и создайте новый объект класса `ClickerState`.
2. Реализуйте цикл, который будет выполняться, пока время в объекте ClickerState не достигнет заданного времени моделирования.
   Внутри цикла реализуйте следующие действия:
   a) Вызовите функцию-стратегию с соответствующими аргументами, чтобы определить, какое строение купить следующим.
      Если стратегия вернёт значение `None`, прервите цикл с помощью оператора `brake`.

   b) Определите, сколько времени должно пройти до того момента, когда строение можно будет купить. Используйте соответствующий метод класса `ClickerState`.
      Если расчётное время превысит время моделирования, прервите цикл.
   c) Подождите, пока покупка станет возможна. Для этого опять следует использовать метод класса `ClickerState`.
   d) Совершите покупку с помощью метода `buy_item()` класса `ClickerState` и обновите стоимость строения в классе `BuildInfo`.
3. После цикла проверьте, осталось ли время до конца симуляции. Если да, то выполните функцию ожидания, чтобы учесть число произведённых за это время печенек.
4. Верните объект класса `ClickerState`, описывающий состояние игры на конец моделирования.

Обратите внимание, что симулятор не должен допустить покупку в том случае, если до окончания времени моделирования у игрока не наберётся достаточно печенек для её оплаты. Однако покупка в последний момент должна быть разрешена.

После реализации класса ClickerState и функции `simulate_clicker()` вы можете проверить работу программы с предоставленной стратегией `strategy_cursor()`. Если всё было реализовано правильно, моделирование игры с настройками по умолчанию должно закончиться в следующем состоянии:

- время: 10000000000.0;
- число печенек у игрока: 6965195661.5;
- скорость производства: 16.1;
- общее число печенек: 153308849166.0.

Этап 3. Стратегии

На последнем этапе следует реализовать следующие функции стратегий:

1. `strategy_cheap()` – стратегия должна выбирать самое дешевое строение, которое можно купить за оставшееся время;
2. `strategy_expensive()` – стратегия должна выбирать самое дорогое строение, которое можно купить за оставшееся время;

3. `strategy_best()` – самая хорошая стратегия, которую сможете придумать.

Если в оставшееся время нельзя купить ни одного строения, функция стратегии должна вернуть значение None. Иначе стратегия должна вернуть строку с корректным именем строения. Возврат None должен привести к завершению цикла моделирования, как было рассмотрено в описании второго этапа.

При реализации каждой стратегии действуйте следующим образом:

1. Напишите код функции стратегии.
2. Добавьте в функцию `run()` соответствующий вызов функции `run_strategy()`.
3. Проверьте корректность работы стратегии. В случае необходимости внесите исправления.

Раскомментируйте код рисования графиков в функции `run_strategy()` для визуализации работы симулятора.

Оценка задания

Реализация класса для хранения состояния игры – 20 %.

Реализация симулятора – 60 %.

Реализация «дешевой» и «дорогой» стратегии – 20 %.

Дополнительные баллы

Реализация стратегии, обеспечивающей производство более 1.25×10^{18} печенек – 15 %.

11. Сим

Постановка задачи

Игра «Сим» изобретена в 1969 году Густавом Симмонсом, известным американским криптографом.

В игру можно играть на листе бумаги. Вначале на бумаге по кругу ставится несколько точек, обычно 6.

Затем игроки ходят по очереди. Каждый ход состоит в закрашивании линии, соединяющей две точки, в цвет игрока. В задании будут использованы красный цвет для игрока, который ходит первым, и зелёный – для второго. Перекрашивать уже покрашенные линии и пропускать ход нельзя. Проигрывает тот игрок, после хода которого образуется треугольник со сторонами, проведёнными этим игроком (учитываются только треугольники, вершинами которых являются исходные точки поля; треугольники, образовавшиеся при пересечении линий внутри окружности, не учитываются).

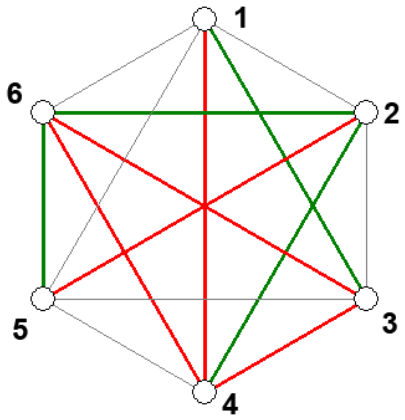


Рис. 11.1 Игра Сим: «красный» игрок проиграл, построив треугольник 3–4–6

Эта игра является результатом исследования математической теории, известной под названием теории Рамсея. В математической постановке задачи точки называют **вершинами** (англ. vertex), а соединяющие их линии – **рёбрами** (англ. edge). Совокупность рёбер и вершин называют **графом** (англ. graph) [12].

С помощью методов теории Рамсея доказано, что в игре Сим с 6 вершинами невозможна ничья, так как любая двухцветная раскраска полного

графа[9] с 6 вершинами обязательно включает треугольник одного цвета. Это свойство может не выполняться для другого числа вершин.

Известно, что для этой игры существует выигрышная стратегия, однако пока не удалось сформулировать её в такой форме, которую было бы удобно запоминать человеку.

Познакомиться с математическими свойствами игры можно в работе [13].

Целью задания является разработка компьютерного игрока в игру Сим, выбирающего ходы с помощью метода Монте-Карло [14].

Предоставляемый код

Для разработки кода следует использовать шаблон sim_template.py, который содержит заголовки функций, которые необходимо разработать, и вызов модуля интерфейса.

Вместе с шаблоном предоставляются также файлы, содержащие код, который следует использовать в решении:

- sim_board.py – реализация класса игрового поля;
- sim_gui.py – графический интерфейс игры;
- sim_compare.py – функции для сравнения эффективности стратегий.

Файл sim_board.py

Файл sim_board.py содержит класс `SimBoard`, который реализует игровое поле игры Сим, функцию `other_player()` и ряд констант.

Константы `PLAYER_RED` и `PLAYER_GREEN` используются как идентификаторы первого и второго игроков, соответственно.

Набор констант, начинающихся на `STATE_`, используется для описания состояния игры:

- `STATE_PLAY` – идёт игра, у следующего игрока есть возможность сделать ход;
- `STATE_DRAW` – игра закончена ничьёй;
- `STATE_RED` – игра закончена победой первого (красного) игрока;
- `STATE_GREEN` – игра закончена победой второго (зелёного) игрока.

[9] Полным называют такой граф, в котором существуют рёбра между всеми парами вершин. В игру Сим обычно играют именно на полном графе, хотя можно использовать и не полные графы.

Обратите внимание, что значения двух последних констант соответствуют идентификаторам игроков PLAYER_RED и PLAYER_GREEN. Это позволяет определить, выиграл ли тот или иной игрок, используя сравнение состояния игры с идентификатором игрока.

При использовании этих констант в коде следует ссылаться на них по именам, а не прописывать численные значения. Для того чтобы имена можно было использовать из других файлов, следует вначале импортировать файл sim_board, а затем ссылаться на эти константы с префиксом sim_board, как показано в следующем примере:

```
import sim_board      # Импортируем файл.
          …
board = sim_board.SimBoard () # Создаём объект игрового поля
          …
if board.get_state() == sim_board.STATE_DRAW    # Проверка, что игра
                                                # закончилась вничью.
```

Функция other_player(player) возвращает «другого игрока»: если на вход подать PLAYER_RED, она вернёт PLAYER_GREEN и наоборот. Эта функция может использоваться для задания очерёдности ходов или узнавания идентификатора оппонента в стратегии. Обращаться к ней из других файлов также следует с префиксом: sim_board.other_player().

Класс SimBoard реализует игровое поле, обеспечивает хранение состояния игры и функции для доступа к нему.

В таблице 11.1 представлено описание доступных для использования методов этого класса.

Таблица 11.1. Методы класса SimBoard

Метод и параметры	Описание
__init__(size) size – размер игры (число точек).	Создаёт поле для игры с заданным числом точек.
__str__	Возвращает строку с описанием состояния игры.
copy()	Возвращает независимую копию поля.
get_size()	Возвращает размер игры.
get_state()	Возвращает состояние игры (значение одной из констант STATE_...).
get_current_player()	Возвращает идентификатор игрока, который

	должен сделать следующий ход.
`get_all_edges()`	Возвращает кортеж (тип `tuple`) всех рёбер. Этот кортеж включает как занятые игроками так и свободные рёбра, и всегда одинаков для игры одинакового размера.
`get_free_edges()`	Возвращает множество (тип `set`) свободных рёбер. Это множество следует использовать в стратегии при выборе следующего хода.
`edge_color(edge)` `edge` – ребро (кортеж).	Возвращает идентификатор игрока, которому принадлежит ребро `edge`. Если ребро свободно, возвращается `None`.
`add_edge(edge)` `edge` – ребро (кортеж).	Делает ход: окрашивает ребро `edge` в цвет игрока, который должен ходить. Возвращает `True`, если ребро окрашено, и `False`, если это нельзя сделать, например ребро уже занято. Распечатывает отладочное сообщение, если ребро задано некорректно. Может приводить к изменению состояния игры.

Вместе с этими методами в классе `BoardState` есть также метод `check_winner()`, который используется методом `add_edge()` для определения, не завершилась ли игра. Вызывать этот метод не следует.

Для представления рёбер класс `BoardState` использует кортежи (тип `tuple`) из двух значений – номеров вершин, которые соединяет ребро. Вершины нумеруются с 0 до номера, на 1 меньшего размера игры. Обратите внимание, что одно и то же ребро может быть описано двумя разными кортежами, в которых вершины переставлены местами (например `(0, 1)` и `(1, 0)`). Методы класса `BoardState` всегда возвращают кортеж, в котором номер первой вершины меньше номера второй. Метод `add_edge()` корректно обработает ребро, указанное в любом порядке. Ошибкой является указание ребра, в котором оба номера вершины совпадают (например, `(4, 4)`) или хотя бы один из номеров не является доступным для данной игры номером вершины.

Файл sim_gui.py

В файле sim_gui.py реализован графический интерфейс игры. Этот класс содержит функцию `run_gui()`, класс `SimGUI` и ряд констант, определяющих внешний вид игры.

Функция `run_gui()` запускает интерфейс. Её параметрами являются `game_size` – размер игры и `machine_player` – функция стратегии. Для запуска игры следует подключить файл sim_gui с помощью команды `import` и вызвать функцию `run_gui()`:

```
import sim_gui
    …
sim_gui.run_gui(6, mc_player)
```

Класс `SimGIU` реализует сам интерфейс. Вызывать методы этого класса в ходе решения не потребуется. С методами этого класса можно познакомиться по комментариям в самом файле.

После запуска интерфейса пользователь может сделать ход, последовательно щёлкнув мышью по вершинам, которые он хочет соединить. Выделенная первым щелчком пользователя вершина подсвечивается цветом игрока. Если пользователь передумал, он может снять выделение повторным щелчком на той же вершине.

После щелчка пользователя по второй вершине игра добавляет соответствующее ребро и отображает ответный ход компьютерного игрока, определённый функцией стратегии, которая была передана как параметр функции `run_gui()`.

Файл sim_compare.py

Чтобы оценить эффективность разных стратегий, можно заставить их играть друг против друга. Файл sim_compare.py содержит функции, которые проводят такие испытания и распечатывают их результаты.

Функция `sim_compare(player1, player2, game_size)` проводит сравнение стратегий `player1` и `player2` на играх размером `game_size`. Она проводит 100 игр (определяется переменной `COMPARE_TRIALS`), в которых за красных играет стратегия `player1`, а за зелёных – `player2`. После проведения игр печатается, сколько раз первая стратегия выиграла, свела игру в ничью или проиграла.

Обратите внимание, что в играх, подобных Сим, важно, какой игрок делает первый ход, поэтому эффективность игры стратегии «за красных» и «за зелёных» может различаться.

В файле sim_compare.py также содержится функция `game()`. Эта функция проводит одну игру-испытание для функции `sim_compare()`. Вызывать её в ходе решения задания не потребуется.

Для вызова функции `sim_compare()` подключите файл sim_compare.py с помощью `import` и используйте префикс `sim_compare.`.

Шаблон sim_template.py

Файл sim_template.py содержит каркас той части приложения, которую следует разработать в ходе решения задачи.

Эта часть включает функцию mc_player(), в которой должна быть реализована стратегия компьютерного игрока на основе метода Монте-Карло, и три вспомогательные функции: `mc_trial()`, `mc_update_scores()` и `get_next_move()`. Работа этих функций будет описана далее.

Этот файл содержит также функцию random_player(), реализующую случайную стратегию. Эту функцию можно использовать для понимания того, как стратегия взаимодействует с остальной частью игры и для сравнения эффективности.

Параметрами функций-стратегий, в том числе функции `random_player()`, являются `board` – игровое поле, представленное классом `SimBoard`, и `player` – идентификатор игрока, ход которого следует оценить.

Случайная стратегия получает от игрового поля информацию о доступных для хода дугах и случайно выбирает одну из них. Выбранная стратегией дуга должна быть возвращена из функции в виде кортежа.

В конце шаблона находится вызов графического интерфейса со случайной стратегией. После реализации стратегии Монте-Карло следует подставить её в этот вызов вместо случайной стратегии.

Стратегия для игры Сим

Стратегия должна использовать метод Монте-Карло для выбора хорошего хода в игре, начиная с текущей позиции. Общая идея метода состоит в том, чтобы смоделировать большое число игр со случайными ходами, начиная с текущей позиции, а затем использовать результаты этих игр, чтобы выбрать рациональный ход.

Рассмотрим одну из смоделированных игр. Если в ней «наш» игрок выиграл, то можно сделать вывод, что рёбра, которые он в ней использовал, являются благоприятными для нас, а те, которые использовал противник – не благоприятными. Напротив, если игра была проиграна, то следует избегать использованных в ней рёбер и предпочесть рёбра противника (чтобы не дать ему их использовать).

В целом получается, что всегда целесообразно выбирать для своего хода те рёбра, которые выбирал выигравший игрок, и не выбирать рёбра проигравшего.

Не забудьте, что окончательный вывод надо сделать не по результатам одной игры, а по результатам множества смоделированных игр. Для этого каждому ребру по результатам каждой игры будут начисляться очки, увеличивающие или уменьшающие степень предпочтительности этого ребра для следующего хода. Полученные в результате накопления на большом числе игр очки следует использовать для выбора следующего хода.

Таким образом, стратегия выбора хода методом Монте-Карло реализуется следующим образом:

1. Начинаем с текущей позиции.
2. Повторять большое число раз:
   a) смоделировать игру со случайным выбором ходов, начиная с текущей позиции и до завершения игры;
   b) скорректировать очки рёбер на основе результата игры.
3. Случайно выбрать ребро с максимальными очками из числа доступных для хода[10].

Наиболее сложным среди этих действий является пункт 2b – корректировка очков. Рассмотрим, как оценить очки для одной игры.

Стратегия должна вычислить очки для каждого ребра. Начисляемые по результатам игры очки зависит от того, кто выиграл рассматриваемую игру.

[10] Одинаковое максимальное значение очков могут иметь сразу несколько рёбер, доступных для следующего хода. В этом случае можно выбрать любое из них – стратегия не может определить, какое из них является лучшим. Возможно, все они действительно являются одинаково хорошими.

Если игра закончилась в ничью[11], то очки не начисляются, так как эта игра не может помочь определить выигрышную стратегию.

Если игра закончилась выигрышем «нашего игрока», то все «наши» рёбра должны получить положительную оценку (равную значению `MC_WIN` в шаблоне), а рёбра оппонента – отрицательную (значение `MC_LOOSE`).

Напротив, в случае проигрыша следует оценить «свои» рёбра значением `MC_LOOSE`, а рёбра оппонента – значением `MC_WIN`.

Не забудьте, что оценка должна вычисляться по результатам всех игр, поэтому полученные за каждую игру очки следует складывать с очками этого ребра, накопленными на данный момент. Начать следует с оценки `0` для всех рёбер.

Рекомендованный порядок выполнения задания

В ходе работы над заданием следует реализовать 4 функции.

Функция `mc_trial()`

Эта функция должна сгенерировать случайную игру начиная из заданного состояния. Единственным параметром функции является `board` – состояние игрового поля в виде класса `SimBoard`.

Реализация этой функции проста – в цикле получайте список свободных рёбер, выбирайте из них случайное и делайте следующий ход. Не потребуется даже отслеживать очерёдность ходов, так как это автоматически реализуется классом `SimBoard`. Игру следует продолжать до тех пор, пока состояние игры не изменится с `STATE_PLAY` на какое-нибудь другое.

Для выбора случайного ребра можно использовать функцию `random.choice()`. Учтите, что эта функция не может работать с множествами (тип `set`). Чтобы её использовать, преобразуйте множество свободных рёбер в список (тип `list`).

Функция `mc_trial()` не возвращает никакого значения. В результате её действий изменяется объект `board`, переданный ей как аргумент.

Функция `mc_update_scores()`

Параметрами функции `mc_update_scores()` являются:

[11] Это возможно для размеров игры, не равных 6.

- `all_edges` – кортеж (тип `tuple`), содержащий все возможные рёбра в игре;
- `scores` – список (тип `list`) текущих очков для каждого ребра;
- `board` – игровое поле на момент окончания игры (объект класса `SimBoard`);
- `player` – идентификатор игрока, с позиции которого оценивается результат игры.

Стратегия хранит информацию об очках каждого ребра в двух структурах данных: кортеж all_edges хранит сами рёбра, а scores – очки за каждое из них соответственно. Например, если `all_edges = ((0, 1), (0, 2), …)` а `scores = [5, -3, …]`, то это значит, что ребро `(0, 1)` имеет оценку $+5$, а ребро `(0, 2)` – оценку -3.

Функция `mc_update_scores()` получает на вход текущие значения очков и должна обновить их по результатам игры, представленной полем `board`.

Во-первых, проверьте, не закончилась ли игра ничьей. В этом случае ничего делать не надо.

Если игра окончилась победой одного из игроков, следует скорректировать очки всех использованных в этой игре рёбер. Правила оценки описаны в разделе «Стратегия для игры Сим». Следует оценить игру, ориентируясь на то, что «наш» игрок – это тот игрок, идентификатор которого передан в параметре `player`.

Функция `mc_update_scores()` не должна возвращать каких-либо значений. Следует обновить очки прямо в параметре `scores`.

Функция `get_next_move()`

Функция get_next_move() должна выбрать следующий ход, ориентируясь на вычисленную оценку. Параметрами этой функции являются:

- `all_edges` – кортеж (тип `tuple`), содержащий все возможные рёбра в игре;
- `scores` – окончательный список (тип `list`) очков для каждого ребра;
- `board` – игровое поле на текущий момент (объект класса `SimBoard`).

В этой функции следует выбрать для хода одно из рёбер, имеющих максимальную оценку.

Начните с получения списка доступных для хода рёбер.

Выберите среди них рёбра с максимальной оценкой. Таких рёбер может быть много.

Случайно выберите и верните одно из этих рёбер.

Функция `get_next_move()` должна вернуть выбранное ей ребро в виде кортежа (тип `tuple`) из двух элементов – начальная и конечная вершина.

Функция `mc_player()`

Эта функция использует три предыдущие для того, чтобы сформировать стратегию. Параметрами функции являются:

- `board` – игровое поле на текущий момент (объект класса `SimBoard`);
- `player` – идентификатор игрока, за которого нужно выбрать ход.

В этой функции должна быть реализована стратегия вычисления рационального хода методом Монте-Карло.

Вначале получите у игрового поля список всех рёбер. Сформируйте список с начальными значениями очков за каждое ребро (начальное значение должно быть 0).

Реализуйте цикл перебора случайных игр. Число итераций цикла должно задаваться значением `MC_TRIALS`, определённым в начале шаблона.

В цикле генерируйте продолжение текущей игры, используя функцию `mc_trial()`, и оценивайте её результат, используя `mc_update_scores()`. Не забудьте, что функция `mc_trial()` модифицирует переданное ей игровое поле, поэтому не передавайте ей непосредственно текущее поле (оно ещё потребуется в своём исходном виде), а сделайте копию с помощью соответствующего метода.

После цикла выберите оптимальный ход с помощью функции `get_next_move()` на основе вычисленных в цикле очков.

Функция `mc_player()` должна вернуть выбранное для следующего хода ребро в виде кортежа (тип `tuple`) из двух элементов – начальной и конечной вершин.

Оценка стратегии

Реализовав стратегию, проверьте её на себе. Для этого, замените передачу случайной стратегии в интерфейс игры в конце шаблона на стратегию Монте-Карло.

Оцените эффективность стратегии в игре со случайной стратегией. Для этого раскомментируйте обращения к функции `sim_compare()`, расположенные перед вызовом интерфейса. Сообщения о результатах сравнения будут напечатаны в консоли перед запуском игры. Корректно реализованная стратегия не должна оставить случайной стратегии почти никаких шансов на победу.

Дополнительное задание

Представленная в шаблоне функция-стратегия random_player() делает действительно случайный выбор. Она может выбрать и ребро, которое немедленно приведёт к проигрышу, даже если у неё есть доступные альтернативы.

Такое поведение, конечно, нельзя назвать разумным. Реализуйте на базе существующей случайной стратегии усовершенствованный вариант, который будет избегать делать проигрышные ходы, если это возможно.

Учтите, что в конце игры вполне возможна ситуация, когда у стратегии не останется возможности сделать ход, который не приведет к немедленному проигрышу. В такой ситуации стратегия должна сделать какой-то ход.

Сравните эффективность этой стратегии с чисто случайной стратегией и методом Монте-Карло. Она должна выигрывать у первой и проигрывать второй.

Оценка задания

Реализация `mc_trial()` – 20 %.

Реализация `mc_update_scores()` – 30 %.

Реализация `get_next_move()` – 25 %.

Выбор фиксированной альтернативы (например, первого или последнего) среди рёбер с максимальной оценкой – -10%.

Реализация `mc_player()` – 25 %.

Дополнительные баллы

Реализация улучшенной случайной стратегии – 15 %.

13. Покер на костях

Постановка задачи

Покер на костях – азартная игра в кости. В неё могут играть два и более человек. Возможна игра в одиночку с целью получения максимального числа очков.

Для игры используются 5 шестигранных костей с числовыми достоинствами от 1 до 6.

Ход игрока начинается с броска всех костей. Затем игрок может сделать ещё два броска, каждый раз выбирая, какие костей он хочет оставить, а какие – бросить повторно для получения определённой комбинации. Игроку не обязательно делать все три броска – он может остановиться и раньше, если доволен полученным результатом.

В зависимости от получившейся комбинации игрокам начисляются очки. Цель игры – набрать максимальную сумму очков.

Игра состоит из двух этапов. На первом этапе игроки должны добиться выпадения трех или более костей одинакового достоинства. На втором этапе игроки должны выполнять различные комбинации (пара, две пары, тройка, пара и тройка и т.п.). В этом задании будет рассматриваться только первый этап игры.

На первом этапе выпавшая комбинация оценивается следующим образом:

- если игрок получил комбинацию из 3 костей одинакового достоинства, он получает 0 очков;
- если получено более 3 костей одинакового достоинства, то каждая дополнительная кость добавляет игроку очки, равные своему достоинству;
- если получено менее 3 костей, то достоинства недобранных костей вычитаются.

Игрок может в любой момент изменить выбор достоинства, по которому оценивается его комбинация. Понятно, что игрок будет выбирать вариант, обеспечивающий ему максимальный результат.

В таблице 13.1 приведены некоторые комбинации и их очки.

Таблица 13.1. Примеры оценки комбинаций на первом этапе игры

Комбинация	Достоинство для оценки	Очки	Комментарий
1, 1, 2, 3, 6	1	-1	Недобор одной единицы.
2, 2, 2, 3, 5	2	0	Набраны три двойки.
3, 3, 3, 3, 4	3	+3	Набраны четыре тройки, за четвертую добавлено 3 очка.
1, 4, 4, 4, 5	4	0	Набраны три четвёрки.
2, 3, 5, 5, 6	2	-4	Недобор двух двоек. Вариант недобор одной пятерки дал бы игроку -5, поэтому ему выгодно играть двойки.
6, 6, 6, 6, 6	6	+12	Набраны пять шестёрок. За две из них добавляются по 6 очков.

Таким образом, перед игроком в ходе каждого хода встаёт вопрос, какие кости оставить, а какие бросить повторно, чтобы увеличить свои шансы набрать больше очков. Те кости, которые игрок оставил и не стал бросать повторно, будем называть **фиксированными**.

Целью задания является разработка программы, которая оценивала бы ожидаемые очки для каждого из возможных вариантов фиксации костей и рекомендовала бы ему оптимальный вариант. Предполагается, что программа используется перед последним броском костей, т.е. необходимо просчитать последствия только одного броска.

Анализ комбинаторной сложности игры

В предыдущем задании для реализации стратегии в игру Сим использовался метод Монте-Карло. В этом методе компьютер случайным образом просчитывает несколько возможных вариантов развития событий и по их результатам оценивает целесообразность ходов. Такой подход даёт приближённое решение, так как в анализе учитываются не все возможные варианты развития событий. Даже если случайный поиск перечислит все варианты (это возможно, когда игрокам остаётся мало возможных ходов), частота появления каждого из них в конкретном случае может не совпасть с точным значением вероятности. Известно, что оценка метода Монте-Карло стремится к вероятности только в пределе и достигнет её лишь при бесконечном числе попыток, что не возможно на практике.

В решении этого задания будет использован тот факт, что в данном случае можно перебрать все возможные варианты развития событий, вычислить их вероятности и в результате точно определить ожидаемые последствия (очки) каждого решения.

Используем методы комбинаторики [15], для того, чтобы проверить, что полный перебор действительно может быть реализован за приемлемое время.

Рассмотрим ситуацию перед последним броском игрока. В этот момент он уже имеет какую-то комбинацию выброшенных достоинств костей. Игроку следует принять решение, какие кости он оставляет, а какие хочет повторно бросить.

Такое решение можно закодировать вектором из 0 и 1: пусть 0 означает, что значение кости фиксируется, а 1 – кость будет брошена заново[12]. При таком обозначении, все возможные решения игрока представляются в виде строк из 5 символов в алфавите $\{0, 1\}$. Как известно из комбинаторики, число таких строк равно $2^5 = 32$. Множество всех доступных для игрока действий называют в математике **множеством всех подмножеств**. Как мы только что убедились, мощность (число элементов) этого множества равна 2^n, где n – число элементов исходного множества.

Отметим, что 32 – это оценка сверху: такое число комбинаций возможно только в том случае, если все достоинства на костях игрока различны. Если есть повторы значений, то число различных решений будет меньше. Допустим, у игрока выпали две единицы. Если он решит одну из них фиксировать, а вторую кинуть ещё раз, то ему всё равно, какая из них какая. Соответственно при наличии повторов число решений игрока будет сокращаться.

После того, как игрок примет решение о фиксации костей, в дело вступает случайность: в результате броска незафиксированные кости принимают новые значения. Число возможных вариантов выпадения костей в данном случае определяется формулой $6^{num\_free\_dice}$, где num_free_dice – число незафиксированных костей. В худшем случае, если все кости кидаются заново, $num\_free\_dice = 5$ и общее число комбинаций равно $6^5 = 7776$. В лучшем случае, если игрок фиксирует все кости, ничего бросать не надо и количество вариантов развития событий равно 1.

[12] Обратите внимание, что в программе такой способ кодирования не будет использоваться.

В качестве грубой оценки сверху можно было бы умножить 32 на 7776 и получить $248\,832$, но эта оценка будет сильно завышена. Чтобы получить более точную оценку, следует по отдельности проанализировать ситуацию для каждого количества зафиксированных костей. Если игрок фиксирует n костей, то бросать он будет $5-n$ костей. Число вариантов выпадения этих костей соответственно равно 6^{5-n}. Число способов зафиксировать n костей равно $\mathrm{C}_5^n = \frac{5!}{(5-n)!n!}$ (из 5 костей выбирается сочетание из n штук). Так как число зафиксированных костей может изменяться от 0 до 5, получаем итоговую оценку:

$$\sum_{n=0}^{5} 6^{5-n} \frac{5!}{(5-n)!\,n!} =$$

$$= 7776 * 1 + 1296 * 5 + 216 * 10 + 36 * 10 + 6 * 5 + 1 * 1 = 16807.$$

Обратите внимание на симметричность последовательности числа способов зафиксировать кости: она сначала возрастает – $1, 5, 10$, а потом убывает – $10, 5, 1$. Число сочетаний всегда ведёт себя подобным образом.

Таким образом, в худшем случае (когда все кости игрока различны) потребуется проанализировать $16\,807$ вариантов развития событий, что совсем не много. Если в наборе игрока есть повторения, число вариантов будет ещё меньше.

Предоставляемый код

Для разработки кода следует использовать шаблон dice_pocker_template.py, который содержит заголовки функций, которые необходимо разработать.

Разрабатываемая рекомендательная система не содержит пользовательского интерфейса, поэтому никакой вспомогательный код не предоставляется. В ходе решения задачи полезно проконсультироваться с примерами из файла enumeration.py.

Рекомендованный порядок выполнения задания

В ходе работы над заданием следует реализовать 4 функции.

Функция `score()`

Эта функция должна оценить руку (набор костей) согласно правилам первого этапа Покера на костях.

Параметром функции является hand – набор достоинств костей, которые выбросил игрок.

Обратите внимание на то, что игрок выбирает для оценки вариант, обеспечивающий ему максимальные очки. Поэтому следует перебрать все возможные достоинства, оценить руку по каждому из этих достоинств и вернуть максимальную из полученных оценок.

Для проверки функции можно использовать комбинации из таблицы 13.1, а также другие комбинации.

Функция expected_value()

Целью этой функции является вычисление оценки одного набора зафиксированных игроком костей – ожидаемого значения [16] очков при данном наборе.

Параметрами функции являются:

- held_dice: фиксированные достоинства, которые игрок оставляет без изменения;
- num_die_sides: число сторон кости;
- num_free_dice: число костей, которые игрок будет бросать.

Для того чтобы оценить полезность фиксации заданного набора костей (`held_dice`), необходимо:

- сгенерировать все возможные варианты выпадения оставшихся костей;
- оценить каждую из получившихся в результате рук (включая фиксированную и «выпавшую» части);
- вычислить ожидаемое значение очков.

Реализация этой функции прямолинейна:

1) сгенерируйте варианты выпадения `num_free_dice` костей, используя функцию `gen_all_sequences()` из примера enumeration.py;
2) с помощью функции `score()` оцените каждую руку, полученную объединением фиксированной части с одним из вариантов;
3) вычислите сумму очков, полученных в пункте 2;
4) поделите сумму на число проанализированных вариантов и верните результат.

Функция gen_all_holds()

Эта функция должна сгенерировать все возможные варианты зафиксировать значения костей так, чтобы в дальнейшем можно было оценить каждый из них.

Параметром функции является hand – набор достоинств костей, которые выбросил игрок.

Результатом работы функции должно быть множество (тип set), состоящее из кортежей (тип tuple). Каждый кортеж включает те достоинства, которые игрок фиксирует перед следующим броском.

Так как множество всех возможных вариантов фиксации достоинств велико, проверку работы функции gen_all_holds() целесообразно начать с уменьшенных вариантов руки. Некоторые из них приведены в таблице 13.2.

Таблица 13.2. Примеры результатов работы функции gen_all_holds()

Рука	Ожидаемый результат gen_all_holds()
(1, 2)	{(), (1,), (2,), (1, 2)}
(1, 1)	{(), (1,), (1, 1)}
(1, 2, 3)	{(), (1,), (2,), (3,), (1,2), (1,3), (2,3), (1, 2, 3)}

В общем случае полученное в результате множество должно включать: пустой кортеж; кортежи, содержащие одно достоинство; кортежи, содержащие пары достоинств; кортежи, содержащие тройки достоинств; и так далее до кортежа, совпадающего с рукой.

Не забывайте, что тип set в Python хранит множество уникальных значений – добавление уже существующих кортежей будут проигнорировано. Например, для руки (1, 1) будет сохранён только один кортеж (1,). Это соответствует требованиям задачи.

Реализация этой функции не потребует написания большого количества кода, однако она потребует внимания и тщательного обдумывания. Структура алгоритма для этой функции может повторять структуру алгоритма функции gen_all_sequences().

Следует реализовать цикл, проходящий по всем достоинствам в руке. Внутри цикла вычисляйте множество всех возможных фиксаций для первых

k достоинств, на основе множества всех фиксаций для $k-1$ достоинства. Процесс построения результата для руки (4, 5, 6) показан в таблице 13.3.

Таблица 13.3. Построение множества всех фиксаций для руки (4, 5, 6)

Шаг цикла (*k*)	*k*-е достоинство	Результирующее множество
Перед началом цикла.		{()}
1	4	{(), (4,)}
2	5	{(), (4,), (5,), (4,5)}
3	6	{(), (4,), (5,), (4,5), (6,), (4, 6), (5, 6), (4, 5, 6)}

Тщательно протестируйте эту функцию, перед тем как переходить к реализации оставшейся части задания.

Функция `strategy()`

Эта функция использует предыдущие для того, чтобы сформировать оптимальную стратегию игры. Параметрами функции являются:

- hand: полная рука игрока (кортеж достоинств выпавших костей);
- num_die_sides: число сторон кости.

Для принятия оптимального решения игрок должен рассмотреть все возможные варианты фиксации костей и выбрать вариант с максимальным ожидаемым значением оценки.

Используя уже реализованные функции, это будет не сложно сделать:

1) сгенерируйте все возможные варианты фиксации, используя функцию `gen_all_holds()`;
2) оцените каждый вариант, используя функцию `expected_value()`;
3) выберите вариант с максимальной оценкой.

Функция должна вернуть кортеж (тип tuple), включающий два элемента:
1) ожидаемое значение очков для оптимального выбора;
2) оптимальный вариант фиксации, в виде кортежа с достоинствами, которые надо зафиксировать.

Дополнительное задание

Проверьте качество реализованной стратегии. Для этого сделайте следующее.

1. Реализуйте функцию `game(strategy)`.
   Эта функция должна сыграть один шаг игры под управлением функции стратегии `strategy()`:
   a) сгенерируйте состояние костей перед броском игрока, используя функцию gen_all_sequences();
   b) получите от стратегии вариант фиксации костей и выполните бросок оставшихся костей;
   c) вычислите и верните оценку итоговой комбинации.
2. Реализуйте функцию `test_strategy(strategy)`.
   Эта функция должна вычислить ожидаемое число очков при игре под управлением заданной стратегии.
   Используя функцию `game()`, организуйте множество игр, вычислите сумму полученных очков и поделите их на число попыток.
3. Реализуйте случайную стратегию для игры «Покер на костях».
   Используя `test_strategy()` сравните её эффективность с оптимальной стратегией, разработанной в основной части задания.

Оценка задания.

Реализация `score()` – 20 %.

Реализация `expected_value()` – 20 %.

Реализация `gen_all_holds()` – 40 %.

Реализация `strategy()` – 20 %.

Дополнительные баллы

Реализация оценки стратегии с помощью `game()` и `test_startegy()` – 10 %.

Реализация случайной стратегии – 5 %.

14. Кошки-мышки

Постановка задачи

В данном проекте вам предстоит смоделировать кошек и мышек, взаимодействующих на сетке. Как принято, кошки голодны и собираются хорошо пообедать. В результате кошки преследуют мышек, а мышки от них убегают. Чтобы сделать модель управляемой, кошки и мышки будут двигаться строго по сетке. В рассматриваемой модели кошки могут за 1 шаг моделирования передвинуться только на одну клетку вверх, вниз, вправо или влево. Чтобы компенсировать отсутствие зубов у мышек, они смогут перемещаться не только по этим четырём направлениям, но и на соседние диагональные позиции. Если кошка ловит мышку, попадая в одну клетку с ней, она вкусно обедает.

Чтобы усилить реальность модели, некоторые из клеток сетки следует пометить как запрещённые для прохода кошек и мышек, так что они не смогут двигаться через эти клетки. В ходе решения задачи предстоит реализовать класс `CatMouse`, который инкапсулирует основные механизмы моделирования и взаимодействует с предоставляемым графическим интерфейсом. Класс `CatMouse` является подклассом класса `Grid` и наследует его методы.

Проходимые клетки соответствуют значению `EMPTY` (пустым ячейкам), а непроходимые – значению `FULL` (занятым ячейкам). Кошки и мышки могут передвигаться только по пустым клеткам сетки. И при этом несколько кошек или мышек могут находиться в одной и той же клетке сетки.

Класс `CatMouse` содержит 2 списка – для кошек и для мышек. Заметим, что записи в каждом из списков являются кортежами вида `(row, col)` (англ. строка, столбец), представляющим позиции кошки или мышки на сетке. На каждом шаге моделирования происходят обновление либо позиций кошек в зависимости от положения мышек, либо позиций мышек, исходя из позиций кошек.

Предоставляемый код

Для разработки кода следует использовать шаблон cat_mouse_template.py.

Вместе с шаблоном предоставляются также файлы, содержащие код, который следует использовать в решении:

- grid.py – реализация класса сетки,
- bfs_queue.py – реализация класса очереди,
- cat_mouse_gui.py – графический интерфейс игры.

Файл grid.py

Файл grid.py содержит класс `Grid` (англ. сетка), реализующий сетку с квадратными ячейками. Этот класс должен являться суперклассом для класса `CatMouse`.

Ячейки сетки индексируются номером строки и номером столбца, как показано на рисунке 14.1. Для ссылок на ячейки будут использоваться кортежи (тип tuple), содержащие номера строки и столбца.

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 0)	(3, 1)	(3, 3)	(3, 4)	(3, 5)

Рис. 14.1. Индексация сетки

Каждая ячейка сетки может быть или занятой (представляется константой FULL), или пустой (константа EMPTY).

Важным отношением между ячейками на сетке является отношение соседства. В рассматриваемой задаче это отношение показывает, на какие ячейки может перейти игровой объект. Различают два вида соседства: 4-соседями являются ячейки, расположенные справа, слева, сверху и снизу от рассматриваемой. В случае 8-соседства к этим 4 добавляются ещё 4 ячейки, расположенные по диагонали. Ячейка и её соседи показаны на рисунке 14.2.

Синий цвет показывает рассматриваемую ячейку, а оранжевый – соседей. Обратите внимание, что ячейки, расположенные по краям сетки имеют меньше соседей. Код класса `Grid` корректно обрабатывает такие ситуации.

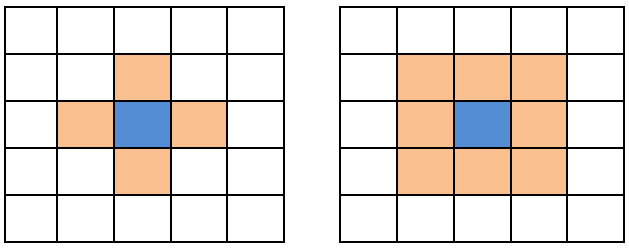


Рис. 14.2 Отношение соседства на сетке (a – 4-соседи, b – 8-соседи)

Методы класса Grid представлены в таблице 14.1.

Таблица 14.1. Методы класса Grid

Метод и параметры	**Описание**
`__init__(grid_height, grid_width)` `grid_height` – высота ячейки; `grid_width` – ширина ячейки.	Создаёт сетку заданного размера (с заданным числом строк и столбцов). Изначально все ячейки пусты (`EMPTY`).
`__str__()`	Возвращает описание сетки в виде текста.
`get_grid_height()`	Возвращает высоту сетки (число строк).
`get_grid_width()`	Возвращает ширину сетки (число столбцов).
`clear()`	Очищает сетку – делает все ячейки пустыми (`EMPTY`).
`set_empty(cell)` `cell` – ячейка.	Делает заданную ячейку пустой (`EMPTY`).
`set_full(cell)` `cell` – ячейка.	Делает заданную ячейку занятой (`FULL`).
`is_empty(cell)` `cell` – ячейка..	Возвращает `True`, если заданная ячейка пуста.
`four_neighbors(cell)` `cell` – ячейка.	Возвращает всех 4-соседей заданной ячейки. Соседи возвращаются в списке (тип `list`) кортежей (тип `tuple`) вида (строка, столбец).
`eight_neighbors(cell)` `cell` – ячейка.	Возвращает всех 8-соседей заданной ячейки. Соседи возвращаются в списке (тип

	list) кортежей (тип tuple) вида (строка, столбец).
get_index(point, cell_size) point – точка cell_size – размер клетки	Возвращает ячейку, в которой находится точка point (заданная в координатах холста), при условии, что каждая ячейка имеет размер cell_size.

Шаблон cat_mouse_template.py уже содержит код для подключения файла grid.py и объявления класса CatMouse как наследника класса Grid.

Файл bfs_queue.py

Файл bfs_queue.py содержит класс Queue (англ. очередь), реализующий очередь для реализации алгоритма поиска в ширину. Очередь реализует алгоритм доступа к элементам FIFO (англ. First In First Out – первый пришел, первый ушел): при запросе на получение элемента возвращается тот, который находится в очереди дольше всего.

Методы класса Queue представлены в таблице 14.2.

Таблица 14.2. Методы класса Queue

Метод и параметры	Описание
__init__()	Создаёт пустую очередь.
__str__()	Возвращает текстовое описание очереди.
__len__()	Обеспечивает возможность использовать функцию len() с очередью. Возвращает число элементов в очереди.
__iter__()	Возвращает генератор, перечисляющий объекты в очереди. Позволяет использовать её в циклах for и функциях, работающих с последовательностями.
enqueue(item) item – новый элемент.	Добавляет элемент item в очередь.
dequeue()	Удаляет из очереди самый старый элемент и возвращает его.
clear()	Удаляет все элементы из очереди.

Для того чтобы использовать класс Queue из других файлов, следует вначале импортировать файл bfs_queue, при создании очереди использовать префикс «bfs_queue.», а затем использовать объект как обычно:

```
import bfs_queue            # импортируем файл.
          …
q = bfs_queue.Queue()       # Создаём объект класса Queue.
q.enqueue(25)               # Помещаем элемент в очередь.
```

Файл cat_mouse_gui.py

В файле cat_mouse_gui.py реализован графический интерфейс игры. Этот класс содержит функцию `run_gui()`, класс `CatMouseGUI` и ряд констант, определяющих внешний вид игры.

Функция `run_gui()` запускает интерфейс. Её параметром должен являться экземпляр класса CatMouse. Для запуска игры следует подключить файл cat_mouse_gui с помощью команды `import` и вызвать функцию `run_gui()`:

```
import cat_mouse_gui
    …
cat_mouse_gui.run_gui(CatMouse(30, 40))
```

Класс `CatMouseGIU` реализует сам интерфейс. Вызывать методы этого класса в ходе решения не потребуется. С методами этого класса можно познакомиться по комментариям в самом файле.

После запуска интерфейса пользователь может мышью расставлять препятствия, кошек и мышек на сетке. Переключение, какой именно объект добавляется, осуществляется кнопкой «Добавление: …». Кнопка «Очистить всё» удаляет все объекты из модели. Кнопка «Кошки преследуют» запускает один шаг движения кошек, а «Мышки убегают» – мышек.

Рекомендованный порядок выполнения задания

Работа над проектом будет разбита на три этапа.

Этап 1. Класс CatMouse

На этом этапе предстоит реализовать основные методы класса `CatMouse`. Заметим, что родительским классом для `CatMouse` является класс `Grid` и он наследует соответственно все методы класса-родителя.

Шаблон содержит реализацию метода `__init__` для класса `CatMouse`. Конструктор принимает на вход два обязательных: `grid_height` и `grid_width` и 3 необязательных[13] аргумента: `obstacle_list`, `cat_list` и `mouse_list`, которые содержат список клеток, в которых изначально находятся препятствия – кошки и мышки соответственно.

[13] Если при объявлении метода класса или обычной функции для его аргумента указано значение, то этот метод можно вызывать без указания значений для этих параметров. В этом случае для них используется то значение, которое указано в заголовке метода.

На первом этапе следует реализовать 7 методов класса `CatMouse`:

1. Реализуйте метод clear(), который должен очистить все клетки сетки, список кошек (self._cat_list) и мышек (self._mouse_list).
   Для очистки сетки используйте метод clear() класса Grid.

2. Реализуйте метод add_cat(cell), который должен добавлять одну кошку в список кошек. Аргументом метода является кортеж, содержащий координаты ячейки с кошкой.

3. Реализуйте метод num_cats(), который возвращает число кошек в списке.

4. Реализуйте метод cats(), который должен являться генератором, перечисляющим всех кошек из списка кошек.
   Генератор обязательно должен выдавать кошек в той последовательности, в которой они были исходно добавлены в модель.

5. Реализуйте метод add_mouse(cell), который должен добавлять одну кошку в список кошек. Аргументом метода является кортеж, содержащий координаты ячейки с кошкой.

6. Реализуйте метод num_mice(), который возвращает число кошек в списке.

7. Реализуйте метод mice(), который должен являться генератором, перечисляющим всех кошек из списка кошек.
   Генератор обязательно должен выдавать кошек в той последовательности, в которой они были исходно добавлены в модель.

После того как эти методы будут реализованы, можно запустить интерфейс игры. Он должен позволить добавлять препятствия (клетки чёрного цвета), кошек (клетки красного цвета) и мышек (клетки зелёного цвета). Клетки, в которых одновременно находятся и кошки и мышки, отображаются фиолетовым цветом.

Этап 2. Вычисление поля расстояний

Этот этап является ключевым этапом проекта. Задачей на этом этапе является вычисление для каждой ячейки сетки расстояния до ближайшей кошки (или мышки). Такой двумерный массив называют полем расстояний. На рисунке 14.3 показан пример поля расстояний до двух мышек, обозначенных зелёными клетками. Поле расстояния должно строиться с учётом перемещения по 4-соседям или 8-соседям и с учётом препятствий.

5	4	3	2	∞	6	5	4	3	4
4	3	2	1	∞	∞	∞	3	2	3
3	2	1	0	1	2	3	2	1	2
4	3	2	1	2	3	2	1	0	1
5	4	3	2	3	4	3	2	1	2

Рис. 14.3. Пример поля расстояний: расстояние до двух мышек

Обратите внимание, что расстояния на поле расстояний растут так же, как при обходе ячеек при поиске в ширину. Поэтому для его построения можно использовать алгоритм поиска в ширину.

Алгоритм поиска в ширину можно было бы использовать и для расчёта расстояния от каждой кошки до каждой мышки. Однако в этом случае расстояния пришлось бы отдельно оценивать для каждой пары «кошка – мышка». Но алгоритм поиска в ширину работает не очень быстро. В целом программа работала бы медленно.

Использование поля расстояний помогает ускорить программу – для его построения достаточно запустить алгоритм поиска в ширину один раз, указав в качестве начальных значений положения всех мышек. Далее можно использовать это поле для того, чтобы выбрать направление движения для всех кошек – им нужно передвигаться в те ячейки, расстояние в которых меньше.

Аналогично, построив второе поле расстояний от всех кошек можно определить направление движения для всех мышек – они должны перемещаться в клетки с большим расстоянием.

Таким образом, для перемещения одной категории персонажей достаточно выполнить алгоритм поиска в ширину только один раз.

Рассмотрим алгоритм поиска в ширину [17]:

```
граница ← очередь, инициализированная списком начальных точек
посещённые ← пустое множество
пока граница не пуста:
    текущая_ячейка ← получить ячейку из очереди граница
    для всех соседей текущей_ячейки:
        если сосед не входит в посещённые:
            добавить соседа в посещённые
            добавить соседа в очередь граница
```

Этот алгоритм может быть модифицирован для расчёта поля расстояний для рассматриваемой задачи. Это можно сделать следующим образом:

1. Создайте новую сетку `visited` (англ. посещённый) того же размера, как и исходная сетка. В начале работы алгоритма её ячейки должны быть пустыми.

2. Создайте двумерный список `distance_field` тех же размеров, что и исходная сетка. Всем ячейкам этого списка присвойте значение, равное произведению ширины сетки на её высоту (это значение гарантировано больше любого из расстояний на сетке).
   Вы можете использовать для создания такого массива следующий код:
   ```
   distance_field = [[max_dst for col in range(width)]
                                  for row in range(height)]
   ```
   где `width` - ширина сетки, `height` – высота сетки,
   `max_dst = width * height`.

3. Создайте очередь `boundary` (англ. граница), которая является копией списка персонажей (кошек или мышек). Для реализации очереди используйте класс Queue из файла bfs_queue.py.
   Для всех клеток из очереди присвойте в `distance_field` значение `0` и отметьте эти ячейки как занятые (`FULL`) в сетке `visited`.

4. Реализуйте алгоритм поиска в ширину, как описано выше. При этом,

   a) выбирая соседей, учитывайте, какой вариант соседства используется;

   b) при обработке каждого соседа проверяйте, что эта клетка является пустой в исходной сетке; для заполненных клеток ничего делать не следует;

   c) добавляя соседа в список `boundary`, обновите также расстояние до него в `distance_field`: это расстояние должно равняться расстоянию до текущей ячейки + 1. Соответствующий код может выглядеть, например, так, как показано ниже:
   ```
   distance_field[neighbor[0]][neighbor[1]] =
                       distance_field[cell[0]][cell[1]] + 1
   ```

Алгоритм построения поля расстояний должен быть реализован в методе `compute_distance_field(entity_type)` класса `CatMouse`. Параметр

`entity_type` задаёт, кого из персонажей использовать как начальные точки для алгоритма поиска в ширину. Если этот параметр имеет значение `CAT`, следует использовать кошек, а если `MOUSE` – мышек. При этом в первом случае алгоритм должен использовать 8-соседей, а во втором – 4-соседей[14]. Вам потребуется реализовать соответствующие условия для выбора входных значений в начале алгоритма.

Метод compute_distance_field() должен вернуть вычисленное поле расстояний в виде двухмерного списка.

Для отладки этого метода печатайте полученное поле расстояний в консоль перед возвращением результата. Метод запускается при нажатии на кнопки «Мышки убегают» и «Кошки преследуют» в интерфейсе.

Если при реализации этого метода возникнут трудности, можно обратиться к модели лесного пожара (файл wildfire.py), функция `update_boundary()` из которого является реализацией одного шага поиска в ширину.

Этап 3. Перемещение персонажей

На заключительном этапе реализуются два метода, которые обновляют положение кошек и мышек на сетке.

1. Реализуйте метод `move_cats(mice_distance)`, который обновляет позицию кошек. Его параметром является поле расстояний до мышек `mice_distance`, построенное с помощью `compute_distance_field()`. Необходимо рассмотреть каждую кошку, проверить её 4-соседей и переместить в ячейку, в которой значение поля расстояний является минимальным.

   Обратите внимание, что кошка может остаться на месте, если значения поля в соседних ячейках больше, чем в той, где она находится.

   Кошка не должна переходить в ячейки, закрытые для перемещения («занятые» в сетке модели). Если метод compute_distance_field() будет корректно реализован в соответствии с рекомендациями, то это никогда не произойдёт, так как значение поля расстояний для занятых ячеек всегда больше, чем для свободных.

[14] Если в качестве начальных точек используются положения мышек, значит, поле расстояния будет использовано для перемещения кошек. Так как кошки могут перемещаться только по 4-соседям, то и расстояния для них должны быть посчитаны по 4-соседству.

Если имеются несколько ячеек с одинаковым значением поля, выбирайте ход случайно.

2. Реализуйте метод `move_mice(cat_distance)`, который перемещает мышек на основании поля расстояний до кошек `cat_distance`.
   Этот метод похож на предыдущий. Отличия состоят в том, что мышки должны двигаться в сторону увеличения поля расстояний и рассматривать в качестве кандидатов 8-соседей.
   Обратите внимание, что если кошки автоматически избегали закрытые для прохода клетки, так как значение поля в них было велико, то мышки, наоборот будут стремиться попасть в такие клетки. Поэтому следует отдельно проверять ход на допустимость.

Каждый метод следует проверять после реализации. Эти методы вызываются нажатием на кнопки «Кошки догоняют» и «Мышки убегают» соответственно.

Дополнительное задание

Разработанная в первой части задания модель управляется вручную: для перемещения кошек и мышек нужно нажимать кнопки, при этом можно подыгрывать персонажам. Чтобы сделать модель более реалистичной, необходимо чтобы эти действия выполнялись автоматически, с некоторым временным интервалом.

1. Создайте кнопки «Запустить симуляцию» и «Остановить симуляцию». Обработчики этих кнопок могут создавать/удалять таймер или устанавливать глобальную булеву переменную.

2. Реализуйте обработчик таймера, который будет по каждому нечётному срабатыванию вызывать `move_mice()`, а по чётному – `move_cats()`. Настройте интервал срабатывания таймера так, чтобы симуляция хорошо смотрелась.

3. Модифицируйте обработчиков кнопок «Мышки убегают» и «Кошки догоняют» так, чтобы они не выполняли никаких действий во время работы симуляции.

Оценка задания

Реализация первого этапа	– 20 %.
Реализация вычисления поля расстояний	– 40 %.
Реализация перемещения персонажей	– 30 %.
Персонажи могут заходить на закрытые для прохода ячейки	– -10 %.
Автоматическая симуляция модели во времени	– 15 %.

15. Фракталы

Постановка задачи

Фракталы

Фракталами называют математические множества, обладающие свойством самоподобия: любая часть фрактала подобна всему фракталу целиком. Термин «фрактал» был введён Бенуа Мандельбротом в 1975 году и получил широкую известность с выходом в 1977 году его книги «Фрактальная геометрия природы» [18]. Однако многие фракталы были известны и ранее.

В этом задании требуется разработать программу, которая строит изображения нескольких фракталов. Будут использованы фракталы, построение которых естественным образом осуществляется с помощью рекурсивных процедур.

Первым фракталом будет фрактальное дерево, которое иногда также называют Пифагоровым деревом. Первые итерации построения этого фрактала показаны на рисунке 15.1.

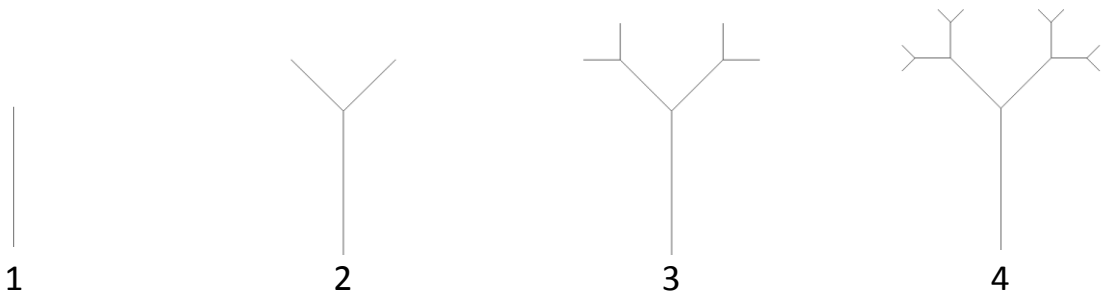


Рис. 15.1. Первые итерации построения фрактального дерева

Процесс начинается с рисования вертикальной линии определённой длины. Затем к верхнему концу этой линии добавляют две линии меньшей длины, направленные под углом 45° направо и налево от направления исходной линии. И далее процесс продолжается – на концах каждой из линий добавляют ещё две линии и так далее. Теоретически, такой процесс построения фрактала должен продолжаться бесконечно. На практике его обычно ограничиваются конечным числом этапов.

Фрактальное дерево является примером класса фракталов, которые называют L-системами, или системами Линденмайера, по фамилии открывшего их в 1968 году венгерского ботаника. Линденмайер разработал

теорию L-систем в ходе изучения роста и развития растений [19]. Многие из фракталов этого класса похожи на реальные растения, их часто применяют для создания изображений растений (рис. 15.2).

Рис. 15.2. Пример изображения растения, сгенерированного с помощью L-системы

Другим классическим примером фрактала из класса L-систем является кривая Коха (рис. 15.3).

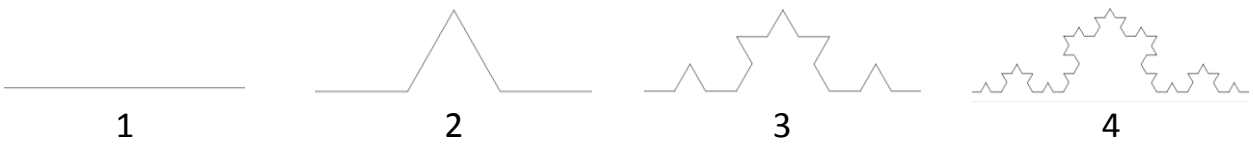
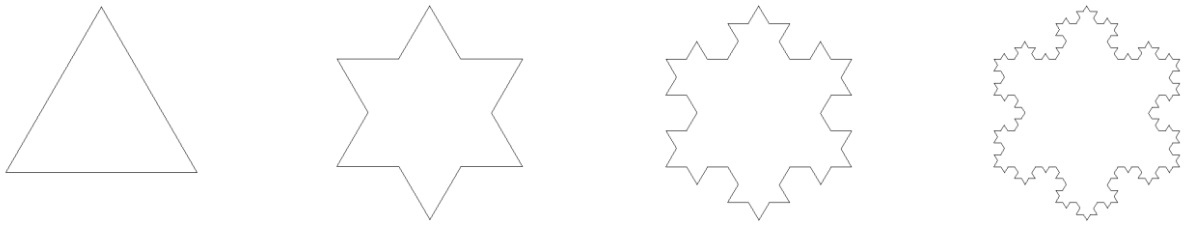


Рис. 15.3. Первые итерации построения кривой Коха

Для построения этой кривой следует взять отрезок и заменить его центральную треть ломаной, повторяющей форму равностороннего треугольника. Затем, каждый из получившихся отрезков подвергается такому же преобразованию.

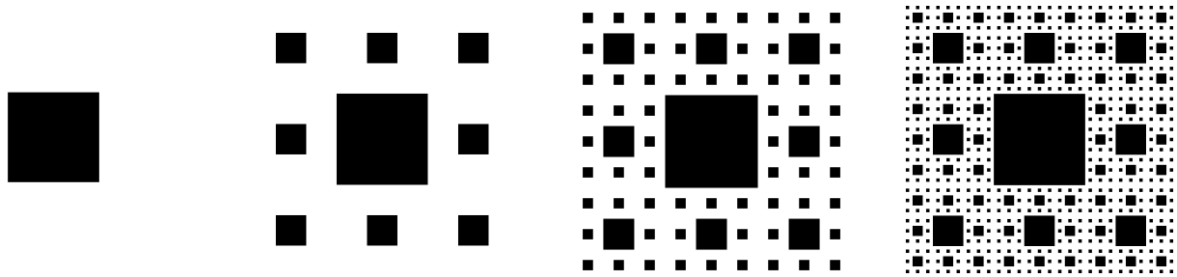
Если начать построение не с отрезка, а с равностороннего треугольника, то получится фигура, известная как снежинка Коха (рис. 15.4).



1 2 3 4

Рис. 15.4. Первые итерации построения снежинки Коха

Последний фрактал не будет относиться к классу L-систем. Этот фрактал известен как ковёр Серпинского (рис. 15.5).



1 2 3 4

Рис. 15.5. Первые итерации построения ковра Серпинского

Построение Ковра Серпинского начинается с квадрата. На первом шаге квадрат разбивается сеткой 3×3 ячейки и центральная ячейка удаляется (или закрашивается). Затем процесс рекурсивно повторяют для 8 оставшихся ячеек.

Ковёр Серпинского, так же как и фракталы из класса L-систем, используют для создания компактных, но эффективных антенн (рис. 15.6).

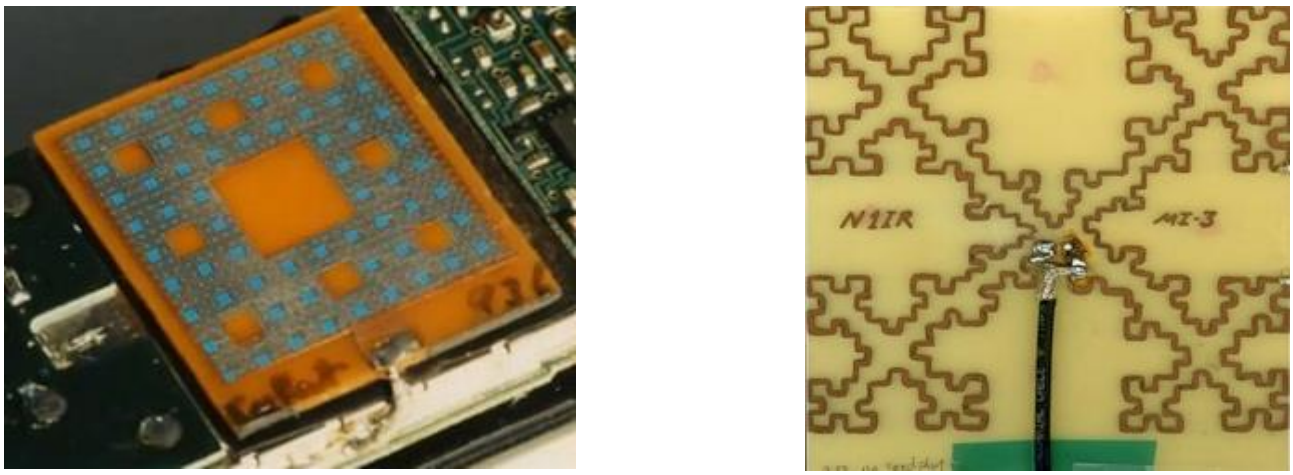


Рис. 15.6. Фрактальные антенны

Архитектура программы

Обратите внимание, что алгоритмы построения этих фракталов рекурсивны: они состоят из отдельных итераций, на каждой итерации могут быть нарисованы какие-то «обычные» компоненты (линии, квадраты, …), а затем осуществляются рекурсивные вызовы для построения оставшихся частей фигуры. При рекурсивном вызове размер фигуры уменьшается, могут измениться её положение и ориентация.

В теории процедура построения фрактала бесконечна. На практике бесконечное вычисление невозможно, поэтому ограничиваются несколькими шагами вглубь рекурсии. На последнем этапе вместо рекурсивного вызова выполняется рисование базовой фигуры, в рассматриваемых фракталах это те же самые «обычные» компоненты, которые используются на предыдущих шагах рекурсии (можно также использовать объект, получаемый на втором шаге рекурсии).

Сама структура рекурсивного алгоритма рисования фракталов одинакова для всех. Различными являются:

- расположение и ориентация начального объекта;
- базовая фигура;
- детали рекурсивного шага: где нарисовать базовые компоненты, а где рекурсивно продолжить построение.

Конечно, можно нарисовать каждый фрактал с помощью одной рекурсивной функции. Но при этом придётся каждый раз повторить те действия, которые являются общими для всех. Избежать такого повторения в данном случае можно, используя наследование. Для этого следует разбить процедуру рисования на небольшие действия и те из них, которые окажутся одинаковыми, поместить в суперкласс. А различающиеся действия поместить в подклассы, создав по одному подклассу для каждого фрактала.

На рисунке 15.7 показана иерархия классов, которые будут использованы в этом задании. Общие для всех фракталов методы и данные вынесены в суперкласс `Fractal`. От этого класса унаследованы классы `CantorSet` (рисует множество Кантора, которое будет рассмотрено в разделе «предоставляемый код»), `FractalTree` (фрактальное дерево), `KochLine` (кривая Коха) и

`SerpinskiCarpet` (ковёр Серпинского). Класс `KochSnowflake` (снежинка Коха) наследуется от класса `KochLine`. Рассмотрим эти классы подробнее.

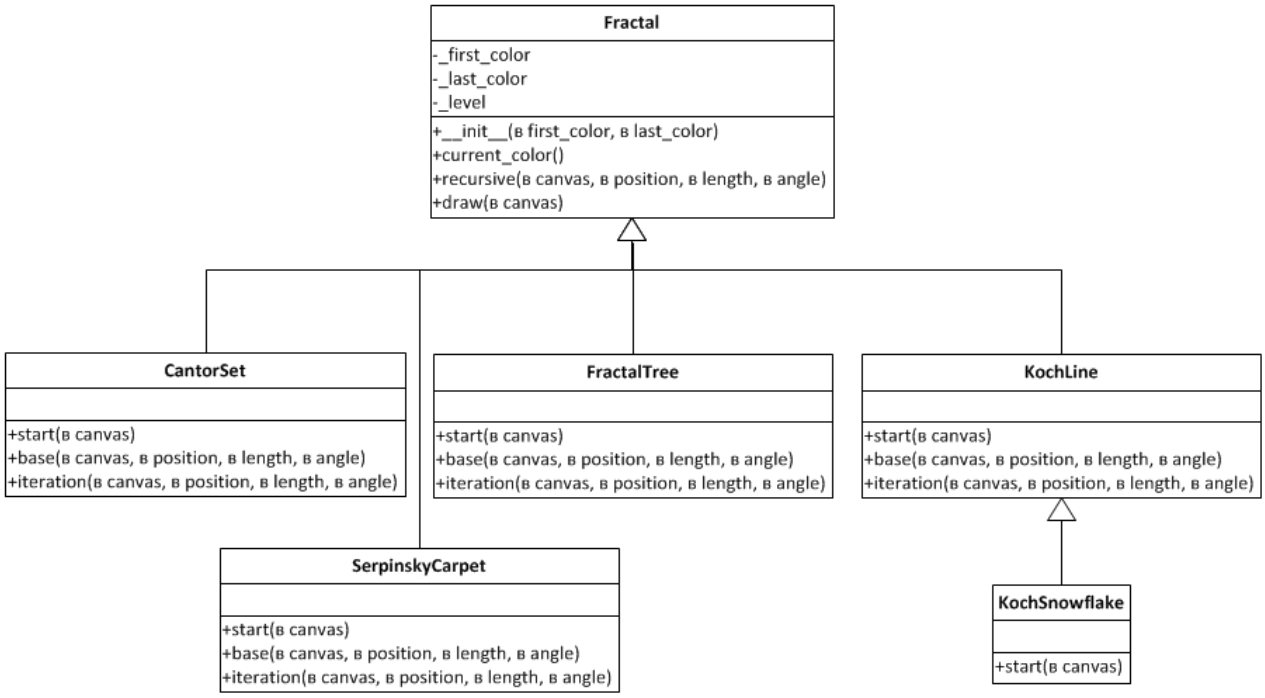


Рис. 15.7. Иерархия классов программы построения фракталов

Так как основной задачей всех классов является рисование фракталов, им не придётся хранить много данных. В данном случае все переменные класса собраны в базовом классе `Fractal`. Переменная `_level` будет хранить текущий уровень рекурсии. Максимальный уровень, до которого будет производиться спуск, будет храниться в глобальной переменной `max_levels`.

Для улучшения эстетических свойств фракталов будем рисовать их с использованием разных цветов для каждого уровня. Переменные `_first_color` и `_last_color` будут хранить цвета для первой и последней итерации, цвета для промежуточных итераций будут вычисляться. Для упрощения вычислений цвет будет храниться в виде кортежа `(R,G,B)`, где R, G и B – значения яркости красной, зелёной и синей компонент.

Конструкторы всех классов будут получать значения цветов для первой и последней итерации. Не забывайте вызывать конструктор суперкласса из конструкторов подклассов и передавать ему эти параметры.

Метод `draw(canvas)` класса Fractal будет начинать процесс рисования. Он будет вызываться из обработчика рисования холста и получит в качестве аргумента холст. Задав начальное значение переменной _level, он вызовет

метод `start(canvas)`, который должен быть определён в подклассе и который должен начать сам процесс рисования.

Метод `recursive(canvas, position, length, angle)` отвечает за рисование рекурсивной части фрактала. Он должен отслеживать изменение уровня рекурсии в переменной _level и в зависимости от него вызывать или метод `iteration()`, или метод `base()` из подклассов. Метод `base()` будет вызываться на последнем шаге и должен нарисовать базовый элемент фрактала. Метод `iteration()` отвечает за реализацию промежуточных итераций: он должен отрисовать фиксированную часть (если она присутствует) и вызвать `recursive()` для рисования «рекурсивных» частей.

Таким образом, в процессе рисования фрактала будет вначале вызван метод `draw()`, затем `start()`, затем `recursive()`, `iteration()`, снова `recursive()`, `iteration()` и так далее, пока не дойдёт до последнего уровня, на котором `recursive()` вызовет `base()`. На рисунке 15.8 показано, какие вызовы будут выполняться для построения трех уровней фрактального дерева.

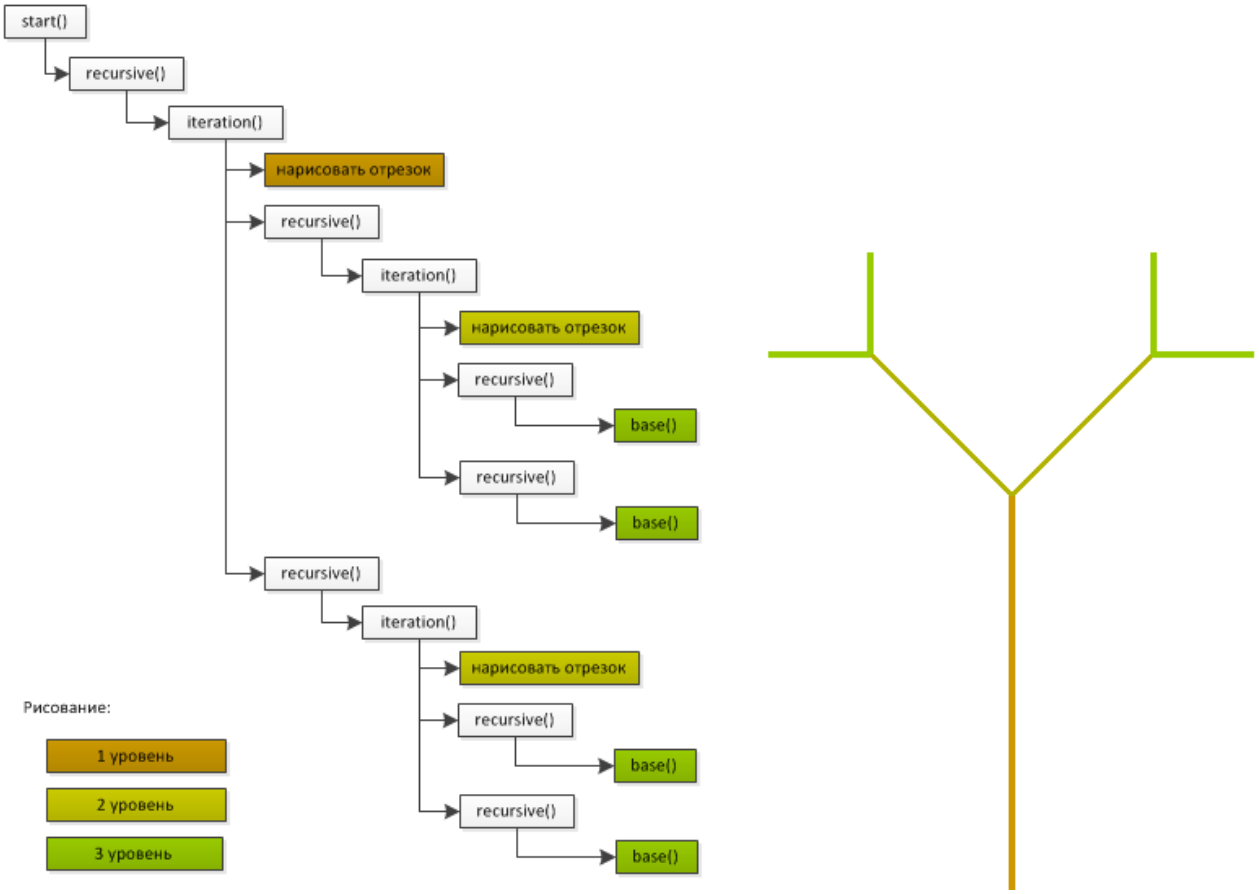


Рис. 15.8. Вызовы методов при построении фрактального дерева до 3-го уровня

Методы `recursive()`, `iteration()` и `base()` получают один и тот же набор аргументов. Первым из них является `canvas` – холст, на котором производится рисование. Оставшиеся три параметра описывают размер и расположение той части фрактала, которую необходимо нарисовать: `position` – начальная точка (кортеж из двух значений `(x, y)`); `length` – длина части и `angle` – угол (направление) в радианах. Этих трех параметров достаточно, чтобы задать положение для всех используемых в задании фракталов (для ковра Серпинского угол не потребуется).

Обратите внимание, что класс `KochSnowFlake` наследуется от класса `KochLine` и переопределяет только один метод – `start()`. Так сделано потому, что снежинка Коха отличается от кривой Коха только начальной фигурой, процедура построения у них одинаковая. Поэтому, сделав класс `KochLine`, можно использовать все его возможности для построения снежинки Коха, заменив только один метод.

Предоставляемый код

Для разработки кода следует использовать шаблон fractal_painter_template.py.

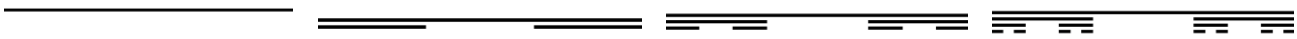
Шаблон содержит заголовки классов Fractal, `FractalTree`, `KochLine`, `KochSnowflake` и `SerpinskiCarpet`.

В шаблоне содержится полная реализация класса CantorSet, которую можно использовать для проверки правильности реализации класса `Fractal`.

В конце шаблона находится реализация графического интерфейса программы.

Класс CantorSet

Класс CantorSet реализует один из вариантов фрактала, соответствующего множеству Кантора [20]. Процесс построения этого фрактала показан на рисунке 15.9.



1 2 3 4

Рис. 15.9. Первые итерации построения множества Кантора

Построение фрактала начинается с горизонтального отрезка. Длина отрезка разбивается на три равные части, под первой и последней из них рисуется следующий уровень фрактала.

Рассмотрим реализацию класса, отображающего этот фрактал.

Конструктор

```
class CantorSet(Fractal):
    def __init__(self, first_color, last_color):
        Fractal.__init__(self, first_color, last_color)
```

Конструктор класса просто передаёт параметры конструктору суперкласса.

Метод start()

```
class CantorSet(Fractal):
    def start(self, canvas):
        self.recursive(canvas, (0, SIZE / 3), SIZE, 0)
```

Метод `start()` отвечает за начало рисования. При выводе этого фрактала позиция (`position`) будет рассматриваться как левая точка отрезка очередного уровня; размер (`length`) – длина этой прямой; угол (`angle`) не будет использован.

Таким образом, данный метод начинает построение фрактала, вызвав рекурсивную процедуру для рисования фрактала, начиная с отрезка шириной с холст (переменная `SIZE` в шаблоне), расположенного на 1/3 высоты холста.

Метод base()

```
class CantorSet(Fractal):
    def base(self, canvas, position, length, angle):
        canvas.draw_line(position, (position[0] + length, position[1]),
                         5, self.current_color())
```

Этот метод отвечает за рисование базового элемента фрактала. В данном случае это просто отрезок прямой. Отрезок начинается с точки `position` и имеет длину `length`.

Для выбора цвета прямой используется метод `current_color()`, реализованный в классе `Fractal`.

Метод iteration()

```
class CantorSet(Fractal):
    def iteration(self, canvas, position, length, angle):
```

```
        self.base(canvas, position, length, angle)
        self.recursive(canvas, (position[0], position[1] + 10),
                       length / 3, 0)
        self.recursive(canvas, (position[0] + 2 * length / 3,
                                position[1] + 10),
                       length / 3, 0)
```

Самый сложный метод этого класса включает три строки и отвечает за реализацию одной итерации построения фрактала. Его работу можно понять, обратившись к рисунку 15.10.

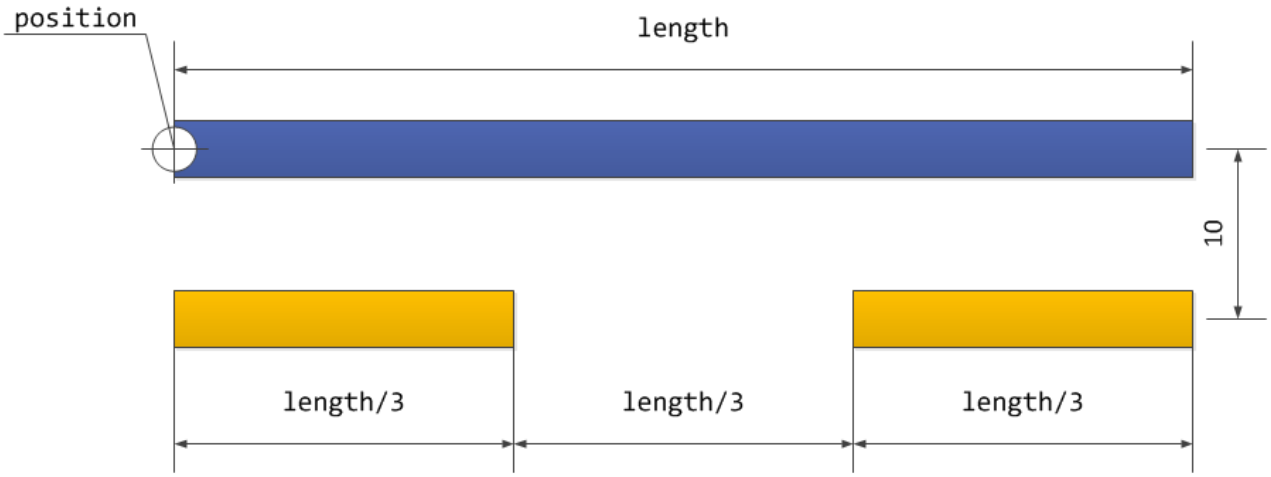


Рис. 15.10. Итерация построения фрактала «множество Кантора»

Итерация начинается с рисования отрезка из точки `position` длиной `length`. Этот отрезок показан синим цветом на рисунке 15.9. Для вывода отрезка используется метод `base()`, так как он строит в точности такой отрезок.

Далее необходимо рекурсивно провести процедуру построения фрактала в местах, показанных (рис. 15.9) жёлтым цветом. Для каждого из них вызывается метод `recursive()` класса `Fractal`. При этом:

- длина отрезка уменьшается в три раза;
- координата по высоте увеличивается (опускается на экране) на 10 пикселей;
- координата по ширине рассчитывается исходя из положения исходного отрезка и размеров частей.

Интерфейс программы

В конце шаблона находится код реализации интерфейса пользователя. Интерфейс включает кнопки для выбора типа фрактала и поле ввода для установки максимального уровня рекурсии. Уровень рекурсии хранится в глобальной переменной `max_levels` и по умолчанию равен 3.

Фрактал для рисования находится в глобальной переменной `f`. Объект, на который указывает эта переменная, пересоздаётся обработчиками кнопок:

- cantor_set() – множество Кантора;
- fractal_tree() – фрактальное дерево;
- koch_line() – кривая Коха;
- koch_snowflake() – снежинка Коха;
- serpinski_carpet() – ковёр Серпинского.

При создании фрактала в каждой функции задаётся своя цветовая схема.

Обратите внимание, что перед созданием окна вызывается функция cantor_set(), соответственно при запуске программа будет отображать множество Кантора.

Обработчик рисования `draw()` вызывает метод `draw()` текущего фрактала: объекта `f`.

Рекомендованный порядок выполнения задания

Работа над проектом разбита на три этапа.

Этап 1. Класс Fractal

На этом этапе предстоит реализовать основные методы класса `Fractal`.

Шаблон содержит реализацию метода `__init__` для класса `Fractal`. Конструктор принимает на вход два обязательных аргумента: цвета для компонентов первого и последнего уровней.

1. Реализуйте метод draw(canvas), который рисует фрактал.
   В этом методе следует установить уровень рекурсии (переменная `self._level`) в `0` и вызвать метод self.start(), который будет определён в подклассах.
2. Реализуйте метод recursive(canvas, position, length, angle).
   В первую очередь этот метод должен увеличить на единицу текущий уровень рекурсии.

Затем проверьте, достиг ли уровень максимального значения, заданного глобальной переменной `max_levels`.

Если нет, следует вызвать метод `self.iteration()` для выполнения очередной итерации. Если максимальная глубина уже достигнута, следует вызвать метод `self.base()`.

Перед выходом из `recursive()` уменьшите уровень итерации на 1.

Если оба метода реализованы верно то при запуске программы будет нарисовано множество Кантора, как показано на рисунке 15.8.

При ошибках вычисления текущего уровня или момента остановки рекурсии программа может зависнуть или нарисовать множество не целиком. Перед продолжением работы убедитесь, что всё работает правильно.

Если множество Кантора в чёрно-белом варианте рисуется правильно, можно добавить к программе цвет. Для этого необходимо реализовать метод `current_color()`. Он должен рассчитать цвет исходя из:

- текущего уровня рекурсии, заданного переменной self._level;
- цвета для первого уровня, заданного переменной self._first_color;
- цвета для последнего уровня, заданного переменной self._last_color.

Цвета для промежуточных уровней будем вычислять по линейному закону, изменяя яркость каждой компоненты от начального значения к конечному пропорционально глубине спуска.

Для этого, вначале вычислите значение alpha так, чтобы оно изменялось от `0` при `self._level==1`, до `1` при self._level==max_level.

Используя alpha, с помощью вспомогательной функции `gradient()` вычислите значения компонент цвета. Помните, что в программе цвет хранится в виде кортежей, первая компонента которых соответствует яркости красного, вторая – зелёного, а третья – синего цвета.

Библиотека SimpleGui использует текстовое представление цвета. Поэтому преобразуйте полученное значение в строку вида «`rgb(R,G,B)`», где `R`, `G` и `B` - вычисленные значения яркости компонент, и верните эту строку.

Если всё было проделано корректно, при запуске программы множество Кантора будет отображено, как показано на рисунке 15.11.

Рис. 15.11. Множество Кантора, окрашенное по уровням рекурсии

Убедитесь, что отрезок верхнего уровня окрашен чистым красным цветом (`rgb(255, 0, 0)`), а отрезки нижнего – чистым зелёным (`rgb(0, 255, 0)`). Если это не так, в первую очередь проверьте правильность вычисления alpha.

Проверьте, что программа работает при числе уровней, равном 1. В этом случае для отображения можно выбрать любой из цветов.

Этап 2. Фрактальное дерево

Завершив работы с суперклассом Fractal, можно переходить к реализации его подклассов, рисующих разные фракталы. Начать лучше с самого простого в реализации – фрактального дерева.

При построении этого фрактала ориентируйтесь на схему на рисунке 15.12.

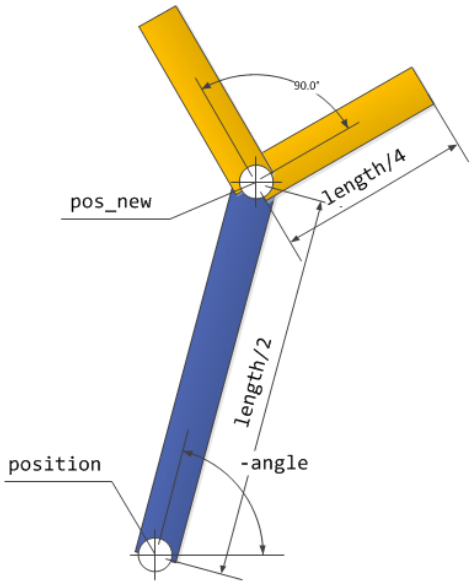


Рис. 15.12. Итерация построения фрактального дерева

В отличие от множества Кантора при построении фрактального дерева ориентация его компонентов будет меняться, поэтому в расчётах будет использован угол, задаваемый аргументом `angle`. Так как в системе координат холста ось ординат направлена вниз, то и направления углов оказываются «перевёрнутыми»: положительное значение угла означает движение по часовой стрелке, а отрицательное – против часовой. Это будет существенно

при ориентации первого компонента фрактала. Далее направления будут вычисляться относительно предыдущего.

Также необходимо помнить, что углы в программе задаются в радианах, 180°=π радиан.

Реализуйте класс FractalTree следующим образом:

1. Реализуйте конструктор класса. Необходимо передать параметры конструктору суперкласса.

2. Реализуйте метод `start()`, который запускает рисование фрактала.
   Рекомендуется расположить начальную точку на середине нижней стороны холста, в качестве размера взять не менее половины размера холста и ориентировать первый сегмент вверх.

3. Реализуйте метод `base()`, который рисует базовую фигуру.
   Этот метод должен нарисовать отрезок, показанный синим цветом на рисунке 15.11. Его начальной точкой является точка `position`.
   Чтобы рассчитать конечную точку (`pos_new`), воспользуйтесь вспомогательной функцией `angle_to_vector(angle, length)`, которая возвращает вектор, ориентированный по направлению `angle` и имеющий длину `length`. Добавьте этот вектор к координатам точки `position`, чтобы получить координаты точки `pos_new`.
   При рисовании линии не забудьте использовать `self.current_color()` для задания цвета.

4. Реализуйте метод `iteration()`, который выполняет одну итерацию. Следуйте схеме, представленной на рисунке 15.11.
   Для рисования синего отрезка используйте метод `base()`.
   Далее два раза вызовите метод `recursive()`, чтобы построить рекурсивные части, обозначенные жёлтым цветом. При этом необходимо скорректировать все параметры расположения частей фрактала:
   - новой позицией будет точка `pos_new`;
   - размер должен уменьшится в два раза;
   - углы для фрагментов вычисляются добавлением и вычитанием 45° (не забудьте перевести в радианы) из текущего угла.

Координаты точки `pos_new` уже вычисляются в методе `base()`, чтобы не вычислять их второй раз, можно вернуть их из метода `base()` и использовать в `iteration()`.

Если всё было выполнено верно, то после запуска программы и нажатия на кнопку «Фрактальное дерево» можно увидеть этот фрактал.

Поэкспериментируйте с пропорциями и ориентацией компонентов фрактала. Ветви дерева не обязательно должны расходиться под одинаковым углом. Можно добавить к дереву и новые ветви. Такие вариации могут порождать различные красивые структуры.

Этап 3. Кривая Коха

Теперь обратимся к следующему фракталу – кривой Коха. Структура итерации для него показана на рисунке 15.13.

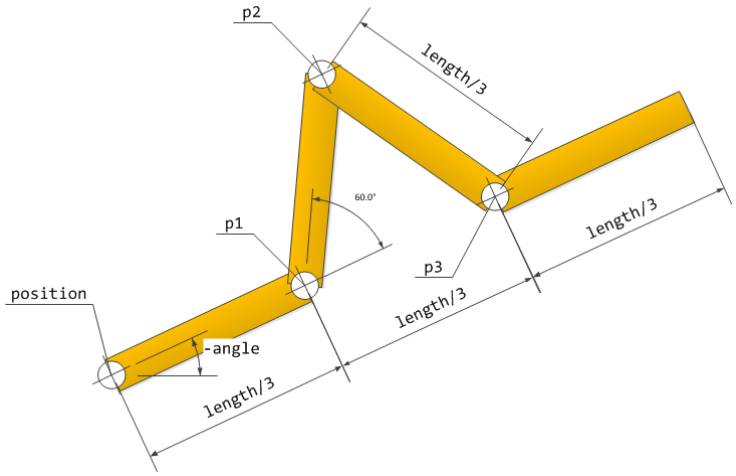


Рис. 15.13. Структура итерации при построении кривой Коха

1. Реализуйте конструктор класса. Необходимо передать параметры конструктору суперкласса.

2. Реализуйте метод `start()`, который запускает рисование фрактала.
   Расположите начальную точку на левой стороне холста в его нижней части. В качестве начального размера используйте полную ширину холста. Начальный угол равен 0 – направление вдоль оси OX.

3. Реализуйте метод `base()`, который рисует базовую фигуру.
   Хотя итерация кривой Коха не содержит никаких «обычных» линий, не забывайте, что основная задача метода `base()` – нарисовать самый нижний из уровней фрактала.

Нарисуйте отрезок, начинающийся в точке `position` и имеющий длину `length` и ориентацию, заданную углом `angle`. Для построения второго конца отрезка воспользуйтесь вспомогательной функцией `angle_to_vector(angle, length)`.

4. Реализуйте метод iteration(), который выполняет одну итерацию. В данном случае необходимо четыре раза вызвать метод `recursive()`. Начальными точками для этих частей будут `position`, `p1`, `p2` и `p3`. Три последние из них можно вычислить используя фунцию `angle_to_vector()` по формулам:

$$p1 = position + angle\_to\_vector(angle, length/3);$$
$$p2 = p1 + angle\_to\_vector(angle - \pi/3, length/3);$$
$$p3 = position + angle\_to\_vector(angle, 2 * length/3);$$

Размер частей в три раза меньше исходного, а ориентация рассчитывается на основании угла angle согласно рисунку 15.13.

Этап 4. Снежинка Коха

Имея реализацию кривой Коха, сделать снежинку Коха не сложно. Так как класс `KochSnowflake` унаследован от `KochLine`, то реализация базиса рекурсии и рекурсивного шага в нём уже есть. Достаточно только реализовать конструктор и новый вариант метода `base()`.

В методе `base()` трижды вызовите `recursive()` для построения частей фрактала, ориентированных по сторонам равностороннего треугольника, расположенного в центре холста. Размер стороны треугольника можно взять равным $^2/_3$`SIZE`.

Дополнительное задание

Используя эту же схему действий, реализуйте рисование ковра Серпинского с помощью класса `SerpinskiCarpet`. Структура итерации для этого фрактала показана на рисунке 15.14.

В этом фрактале базовой фигурой является квадрат, выделенный синим цветом на этом рисунке. Итерация будет состоять из рисования базовой фигуры и восьми рекурсивных вызовов.

Установленный в шаблоне размер холста позволяет отобразить ковёр Серпинского до 5-го уровня рекурсии. Рисование более глубоких уровней будет

работать медленно и потребует увеличения размера холста, чтобы результат можно было увидеть.

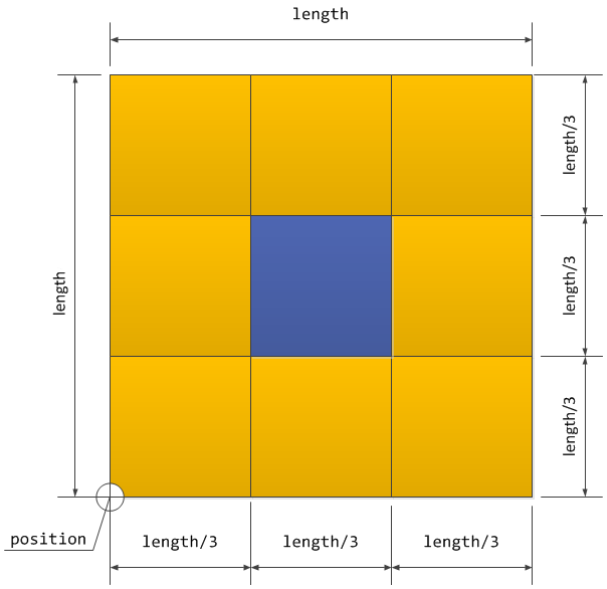


Рис. 15.14. Структура итерации построения ковра Серпинского

Оценка задания

Методы draw() и recursive() класса Fractal – 20 %.

Метод current_color() класса Fractal – 5 %.

Класс FractalTree – 30 %.

Класс KochLine – 35 %.

Класс KochSnowflake – 10 %.

Класс SerpinskiCarpet – 15 %.

Реализация рисования любого из фракталов задания без использования наследования с помощью рекурсивных функций оценивается в 50% от баллов за реализацию соответствующего класса.

Список литературы

[1] *G. van Rossum, B. Warsaw B., N. Coghlan* PEP8 - Style Guide for Python Code [Электронный ресурс]. Дата обновления: 05.07.2001. Режим доступа: http://legacy.python.org/dev/peps/pep-0008/ (дата обращения 26.01.2016).

[2] *Доусон М.* Программируем на Python. М. [и др.]: Питер, 2012.

[3] *Лутц М.* Программирование на Python. СПб.: Символ-Плюс, 2011.

[4] *Бизли Д.* Python. Подробный справочник. СПб.: Символ-Плюс, 2010.

[5] PyCharm [Электронный ресурс] // JetBrains. Режим доступа: http://www.jetbrains.com/pycharm/ (Дата обращения: 26.01.2016).

[6] PyScripter [Электронный ресурс]. Режим доступа: https://code.google.com/p/pyscripter/ (Дата обращения: 26.01.2016).

[7] Spyder [Электронный ресурс]. Режим доступа: https://github.com/spyder-ide/spyder (Дата обращения: 26.01.2016).

[8] *Кормен Т., Лейзерсон Ч. Э., Ривест Р. Л., Штайн К.* Алгоритмы: построение и анализ, 2-е издание. М.: Вильямс, 2005. – 1296 с.

[9] Pong (игра) [Электронный ресурс] // Википедия : электрон. энциклопедия. Режим доступа: https://ru.wikipedia.org/wiki/Pong_(игра) (Дата обращения: 26.01.2016).

[10] *Буч Г., Максимчук Р. А., Энгл М. У., Янг Б.Д., Коннален Д., Хьюстон К. А.* Объектно-ориентированный анализ и проектирование с примерами приложений. М.: Вильямс, 2010.

[11] *Крокфорд Д.* JavaScript: сильные стороны. М. [и др.]: Питер, 2012.

[12] *Ойстин О.* Графы и их применение. М.: КомКнига, 2006.

[13] *W. Slany* Graph Ramsey games [Электронный ресурс]. Режим доступа: http://arxiv.org/abs/cs/9911004 (Дата обращения: 26.01.2016).

[14] *Гмурман В. Е.* Теория вероятности и математическая статистика: учеб. пособие для вузов. М.: Высш. шк., 2003.

[15] *Липский В.* Комбинаторика для программистов под ред. А. П. Ершова. Москва: Мир, 1988.

[16] *Хохлов Ю. С.* Теория вероятностей и математическая статистика [Электронный ресурс] : учеб. пособие, Тверь: Твер. гос. ун-т, 2014.

[17] *Левитин А. В.* Алгоритмы: введение в разработку и анализ. М.: Вильямс, 2006.

[18] *Мандельброт Б.* Фрактальная геометрия природы. М.: Институт компьютерных исследований, 2002.

[19] *P. Prusinkiewicz, A. Lindenmayer* The Algotithmic Beauty Of Plants. New York: Springer-Verlag, 1990 (1996).

[20] *Александров П. С.* Введение в теорию множеств и общую топологию. М., 1977.

[21] Борьба с тридцатилетним багом [Электронный ресурс] // Хабрахабр. Дата обновления: 20.10.2013. Режим доступа: http://habrahabr.ru/post/198174/ (Дата обращения: 26.01.2016).

[22] Excel ошибочно предполагает, что 1900 год – високосный год [Электронный ресурс] // Microsoft, служба поддержки. Дата обновления: 17.12.2015. Режим доступа: http://support.microsoft.com/kb/214326 (Дата обращения: 26.01.2016).

Учебное издание

Сорокин Сергей Владимирович

Введение в программирование на языке Python. Практикум

Учебное пособие

Отпечатано в авторской редакции.
Тверской государственный университет
Факультет прикладной математики и кибернетики
Адрес: 170000, г.Тверь, пер.Садовый, 35