Урок QT. Файлы

Файлы в Python. Типы файлов и работа с ними. Внутреннее устройство файлов

1. Общие сведения о файлах
2. Перевод строки
3. Файловый путь. Относительные и абсолютные пути
4. Кодировки файлов
5. Типичные операции с файлами
6. Чтение файла
7. Запись в файл
8. Закрытие файлов
9. PyQT
10. Важная информация о текстовых файлах.

Аннотация

В уроке даются общие сведения о файлах и их хранении в современных ОС. Затрагиваются наиболее общие аспекты работы с файлами в Python: открытие, чтение/запись, закрытие текстовых и бинарных файлов. Обзорно рассмотрены вспомогательные функции («перемотка», работа с кодировками, построчное чтение).

1. Общие сведения о файлах

Для работы с блоками логически объединенной информации, их хранения, обработки и передачи широко используются файлы.

Файл

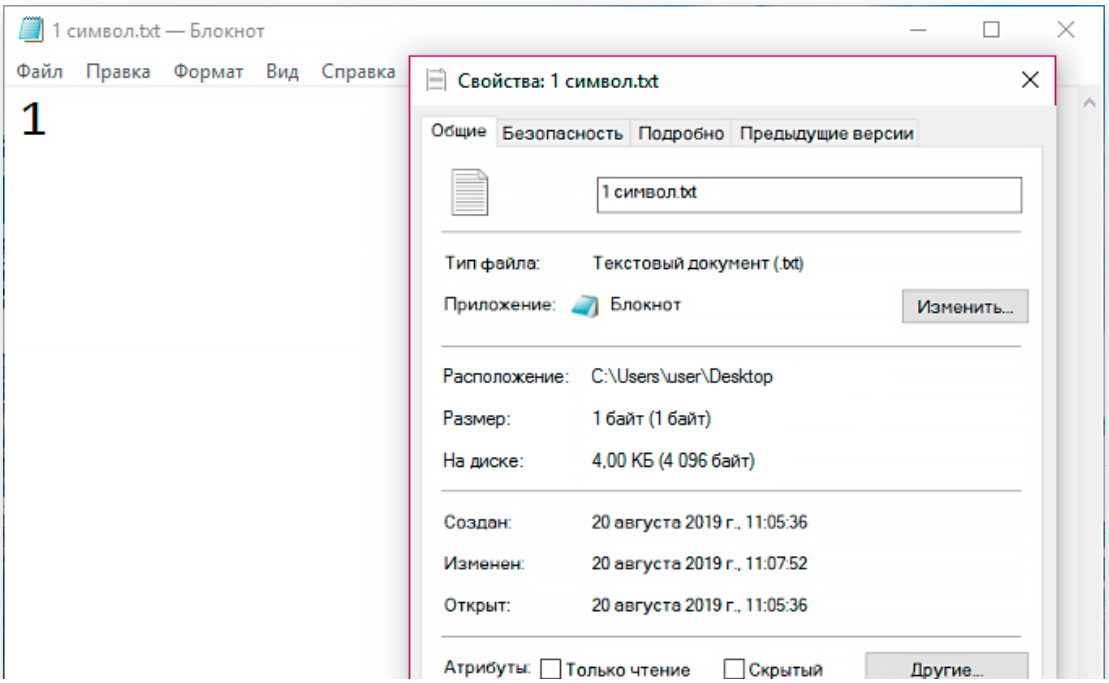
Файл (англ. *file*) — именованная область данных на носителе информации.

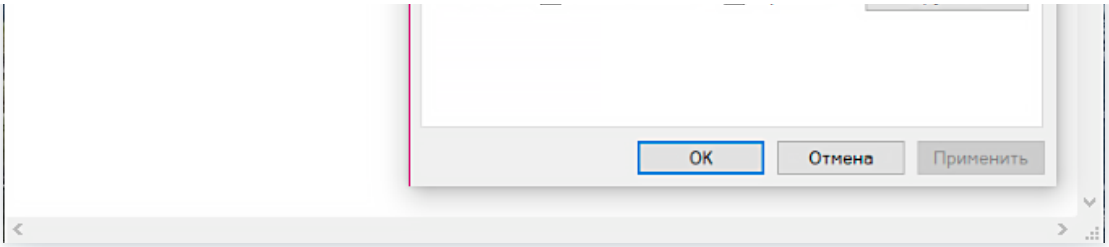
Конкретная физическая организация файлов, их группировка по каталогам (папкам), устройство процедур доступа к информации, механизмы кеширования очень сильно зависят от операционной системы и применяемой в ней файловой системы. Как правило, при работе с файлами прикладные программисты работают на верхнем уровне **абстракции**.

Это означает, что файл представляет просто собой поток байт или текста заданного размера, а его реальное физическое устройство программиста абсолютно не интересует. Но грамотный программист должен хорошо представлять себе некоторые особенности работы с файлами, чтобы не совершать ошибки, связанные с оптимизацией и ускорением работы программы.

О чем надо всегда помнить:

1. Файлы обычно располагаются на носителях, которые работают медленнее, чем оперативная память. Поэтому работа с ними идет в буферизированном режиме. Даже если вы запросите один байт из файла, то считается целый блок (до нескольких килобайт). Затем он переместится в буфер оперативной памяти. Дальше файл читается оттуда, поскольку это быстрее (даже если у вас SSD) и экономит ресурсы чтения/записи внешних носителей. Небольшой файл, к которому часто обращаются, можно (прозрачно для прикладных программ) полностью поместить в оперативную память. На самом жестком диске располагаются буферные зоны с более быстрым доступом — в общем случае мы имеем дело с многоуровневой буферизацией. Поэтому **очень плохо** считывать файл небольшими порциями коротких блоков (до нескольких мегабайт) якобы для уменьшения затрачиваемой памяти.
2. То, как файлы и каталоги (папки) располагаются на жестком диске, зависит от типа файловой системы. Она может поддерживать сжатие, шифрование, разграничение доступа к данным. Никогда файл, кроме совсем маленьких, не размещается полностью и подряд в определенной области диска. Он разбивается на блоки для рационального расходования места. У блоков есть минимальный размер. Поэтому, даже если вы создадите файл из одного байта, он все равно займет целый блок данных (например, 4 Кб).





3. Вся работа по буферизации, чтению, записи, открытию, закрытию файла идет через операционную систему. Чаще всего прикладной программист не работает с файловой системой напрямую.
4. Во многих операционных системах понятие файла как некоторой области данных на носителе, поддерживающих операции чтения/записи, расширено до областей в оперативной памяти, ресурсов сети, оборудования и т. д. Эта концепция называется **«все есть файл»**. Например, в Python есть «файл» sys.stdin, который ассоциирован с клавиатурным вводом и не является классическим файлом.

Есть некоторые различия в именовании файлов в unix-подобных ОС и Windows.

Например, в Linux системные файлы, как правило, не имеют расширения. Все устройства и диски добавлены в общее дерево, корень которого обозначается "/".

Для отделения имен папок используется прямой слеш "/", а не обратный, как в Windows.

Приведем примеры специальных «файлов» в ОС Linux:

- /dev/sd{буква} — жесткий диск (в системах на ядре Linux)
- /dev/sd{буква}{номер} — раздел диска (в системах на ядре Linux)
- /dev/sr{номер} или /dev/scd{номер} — CD-ROM
- /dev/eth{номер} — сетевые интерфейсы Ethernet
- /dev/wlan{номер} — сетевые интерфейсы Wireless
- /dev/lp{номер} — принтеры
- /dev/video{номер} — устройства изображений, камеры, фотоаппараты и т. д.
- /dev/bus/usb/000/{номер} — устройство номер на шине USB первого контроллера (000) (в системах на ядре Linux)
- /dev/tty{номер} — текстовый терминал
- /dev/dsp — звуковой вывод
- /dev/random — случайные данные (псевдоустройство)
- /dev/null — пусто (псевдоустройство)
- /dev/zero — нулевые байты (псевдоустройство)

{номер} — это порядковый номер устройства

2. Перевод строки

Существует два символа `\n` и `\r`, смысл которых взят из эпохи печатных машинок. Посмотрим, какие у них коды:

```
# LINE FEED. Перемещает позицию печати на одну строку вниз
# (изначально – без возврата каретки машинки).
print(ord("\n"))
# CARRIAGE RETURN. Перемещает позицию печати в крайнее левое положение
# (изначально – без перехода на следующую строку).
print(ord("\r"))
```

```
10
13
```

В ОС Windows для перевода строки принимается последовательность `\r\n`, в MacOS (до версии X) — `\r`, а в Linux — `\n`.

Сейчас все чаще во всех ОС используется одиночный `\n`.

Поэтому программисту надо быть внимательным и помнить все варианты перевода строки на той ОС, где будет работать его программа.

3. Файловый путь. Относительные и абсолютные пути

Путь файла (или путь к файлу) — последовательное указание имен каталогов, через которые надо пройти, чтобы добраться до объекта. Каталоги в записи пути разделяются **слешем**. В зависимости от вида ОС слеши могут быть как прямыми, так и обратными.

На ОС Windows путь выглядит так:

```
f = open('C:\\users\\user\\1.txt')
```

Обратный слеш удваивается как служебный символ. Значит, если он нужен сам по себе, его нужно «экранировать». Вспомните, что в языке Python есть служебные символы (`\n`, `\t`).

Для того чтобы сделать работу с файлами универсальнее, в путях файлов в Windows в python-программах рекомендуется ставить прямой слеш. В наших примерах мы так и будем делать.

```
f = open('C:/users/user/1.txt')
```

Если мы укажем относительные пути, например:

```
f = open("user/1.txt")
```

или

```
f = open("./user/1.txt")
```

то Python будет искать файл в каталоге, начав отсчет с папки, в которой находится файл с основной python-программой. Это важно помнить в проектах, где много файлов и происходит импорт модулей или функций.

В относительных путях используют обозначения `"./"` для обозначения текущего каталога и `"../"`

для обозначения родительского каталога или каталога на один уровень выше по отношению к текущему.

```
f = open("../user/1.txt")
```

Здесь Python будет искать файл в каталоге, начав отсчет с папки, в которой находится файл с основной python-программой, затем поднимется на уровень выше, будет искать там папку `"user"`, а уже в ней файл `"1.txt"`

4. Кодировки файлов

Сейчас принято использовать одну из самых распространенных кодировок — **UTF-8**. Мы поступим так же.

UTF-8 — сложная кодировка, в которой символ кодируется от одного до шестью байтами. Подробнее про эту кодировку можно почитать **здесь**.

Но помните, что до сих пор существуют и старые однобайтовые кодировки:

- **Windows-1251**;
- **cp-866**;
- **КОИ-8**.

Они по умолчанию используются в некоторых ОС, например, в Windows.

У пользователей, работающих на компьютерах под управлением ОС Windows, могут возникнуть проблемы с созданием текстовых файлов в кодировке UTF-8. Это происходит из-за того, что ОС Windows до сих по умолчанию использует однобайтовую кодировку Windows-1251. В редакторе Notepad («Блокнот») можно при сохранении указывать ту кодировку, в которой на самом деле хочется сохранить файл, однако все об этом забывают.

Вообще, текстовые файлы стоит открывать в самой IDE PyCharm, что проще всего. При этом нужно помнить, что IDE не добавляет расширение файла автоматически, его нужно указывать явно или выбирать тип из предложенного списка. В этом случае не будет проблем с кодировками. Также можно использовать сторонний текстовый редактор, например, Sublime.

При работе с ОС Linux и MacOS таких проблем не возникает вовсе, поскольку в них кодировка UTF-8 применяется по умолчанию.

Есть еще один вариант, как решить проблему с кодировкой в коде. Надо в первой строке программы написать

```
# -*- coding: utf-8 -*-
```

Это даст интерпретатору указание, что дальнейший код надо воспринимать именно в этой кодировке.

5. Типичные операции с файлами

Все файлы на диске — последовательность байт. Операционной системе все равно, какой смысл у содержимого файла: это видео, чертеж, картинка, текстовый документ и т. д. Все это остается в компетенции прикладной программы.

Единственное исключение сделано для текстовых файлов, потому что их состав максимально прост (до внедрения Юникода одному символу соответствовал 1 байт). Поэтому если программа знает, что файл текстовый, то сразу читает из файла символы, а не поток байт.

Таким образом, все файлы искусственно разделены на **текстовые** и **бинарные**. Но не забывайте, что любой текстовый файл является бинарным.

Так как общение с файлами идет не напрямую, а через ОС, общепринятая последовательность операций с файлом следующая:

1. Попросить ОС открыть файл в различных режимах для чтения или записи, в бинарном или текстовом режиме.
2. Поработать с информацией из файла, используя в том числе операции чтения/записи.
3. Закрыть файл.

После успешного завершения python-программы все файлы закрываются автоматически. Но важно все равно закрывать файл, как только он перестает быть вам нужным. Это поможет избежать конфликтов совместного доступа или риска получить неконсистентный (испорченный) файл, если программа завершится аварийно.

6. Чтение файла

Возьмем файл с **первым томом** «Войны и мира» Льва Толстого.

Для открытия файлов в Python есть функция `open()`.

Выполним `help(open)` для получения справки по этой функции.

```
help(open)
```

```
Help on built-in function open in module io:

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=Tr
    Open file and return a stream.  Raise IOError upon failure.

    file is either a text or byte string giving the name (and the path
    if the file isn't in the current working directory) of the file to
    be opened or an integer file descriptor of the file to be
    wrapped. (If a file descriptor is given, it is closed when the
    returned I/O object is closed, unless closefd is set to False.)

...
```

Из всех параметров мы используем следующие:

- `file` — имя открываемого файла. Оно может быть задано с использованием и относительных и абсолютных путей
- `mode` — режим открытия. `r` (или `rt`) — чтение в текстовом режиме, `rb` — чтение в бинарном режиме, `w` — запись, `wb` — запись в бинарном режиме. По умолчанию файл открывается в режиме `r`
- `encoding` — если работа идет в текстовом режиме, Python должен получить имя кодировки, чтобы корректно работать с данными. Независимо от кодировки файла, в результате чтения будет возвращаться стандартная юникод-строка Python

Пожалуйста, прочтите полное описание параметров в **документации** по функции `open`.

Теперь откроем файл в бинарном режиме и прочитаем первые 20 байт:

```
f = open("files/Толстой.txt", mode="rb")
print(f.read(20))
f.close()
```

```
b'                -- \xd0\x95'
```

Перед строкой стоит модификатор `b`. Он говорит о том, что перед нами поток байт. Поток байт в языке Python представляется классом `bytes`. Если вы открываете файл для чтения в бинарном режиме, результат метода `read()` имеет тип `bytes`.

```
f = open("files/Толстой.txt", mode="rb")
data = f.read()
print(type(data))
print(data[19])
f.close()
```

```
<class 'bytes'>
149
```

Еще раз обратите внимание на то, что в путях до файла используются прямые слеши (/). Можно использовать и обратные, но тогда их придется экранировать либо применять модификатор строки `r`. Кроме того, в unix-подобных ОС принято использовать именно прямой слеш.

Чтобы понять, что делает модификатор `r`, рассмотрим пример:

```
print("ab\n12")
print(r"ab\n12")
```

```
ab
12
ab\n12
```

В первом случае специальный символ `\n` «отработал» и перевел строку, а во втором — вывелся на экран как есть. Модификатор `r` отключает спецсимволы, если он указан перед строкой. То есть каждый символ означает сам себя и — ничего более.

Объект, который возвращает нам функция open, ассоциирован (связан) с открытым файлом и содержит следующие поля и методы:

```
from pprint import pprint
f = open("files/Толстой.txt", mode="rb")
pprint(dir(f))
f.close()
```

```
['__class__',
 '__del__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__enter__',
 '__eq__',
 '__exit__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__lt__',
 '__ne__',
 '__new__',
 '__next__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '_checkClosed',
 '_checkReadable',
 '_checkSeekable',
 '_checkWritable',
 '_dealloc_warn',
 '_finalizing',
 'close',
 'closed',
```

```
 'detach',
 'fileno',
 'flush',
 'isatty',
 'mode',
 'name',
 'peek',
 'raw',
 'read',
 'read1',
 'readable',
 'readinto',
 'readinto1',
 'readline',
 'readlines',
 'seek',
 'seekable',
 'tell',
 'truncate',
 'writable',
 'write',
 'writelines']
```

Напоминаем, что функция `pprint` (pretty-print, «изящный вывод») старается сделать печать больших объектов (списков, словарей) более удобной и читаемой для человека.

Назначение почти всех из них легко определяется из названий.

```
f = open("files/Толстой.txt", mode="rb")
f.read(20)
print(f.name)
print(f.readable())
print(f.tell())
print(f.writable())
print(f.seekable())
print(f.mode)
f.close()
```

```
files/Толстой.txt
True
20
False
True
rb
```

Теперь откроем файл в текстовом режиме и проверим работу некоторых методов:

```
f = open("files/Толстой.txt", encoding="utf-8")
print(f.read(100))
print(f.tell())
print(f.seek(1245))
print(f.read(100))
print(f.tell())
f.close()
```

```
               -- Eh bien, mon prince. Gênes et Lucques ne sont plus que des apanages,
            des
103
1245
ла (грипп  был тогда  новое
          слово, употреблявшееся только  редкими).  В записочках, разосл
1416
```

Что же произошло:

- Если мы открываем файл в текстовом режиме, чтение происходит посимвольно (символ может занимать физически 1, 2, 4 и даже 6 байт в некоторых случаях в зависимости от кодировки). За одну операцию можно прочитать различное количество символов
- Если мы используем бинарный режим, чтение осуществляется побайтно и за одну операцию можно прочитать сразу несколько байт
- Метод `tell` возвращает позицию в байтах от начала файла, а метод `seek` изменяет ее (перематывает) на заданную позицию. Использование `seek` с текстовыми файлами затруднено из-за несоответствия номера байта и номера символа (это видно в предыдущем примере, когда мы считали 100 символов, а `tell` вернул нам смещение 103)

Продолжим экспериментировать:

```
print(f.seek(1246))
print(f.read(1))
```

```
Traceback (most recent call last):
  File "/home/03.py", line 8, in <module>
    print(f.read(1))
  File "/usr/lib/python3.9/codecs.py", line 322, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xbb in position 0: invalid start b
```

Исключение появилось потому, что начиная с 1246-го байта невозможно прочитать символ в кодировке UTF-8.

Из-за буферизации и оптимизации количества операций чтения/записи иногда получается считать вообще весь файл в оперативную память (для этого не надо указывать параметр в методе `read`). Если его размер до десятков мегабайт, это можно комфортно делать. В результате мы получим одну строку

со всем содержимым файла.

```
f = open("files/Толстой.txt", encoding="utf8")
data = f.read()
print(f'Type: {type(data)}, length: {len(data)}')
f.close()
```

```
Type: <class 'str'>, length: 887312
```

В текстовом режиме можно читать файл построчно с использованием метода `readline`, при этом маркером конца строки является символ `\n`:

```
f = open("files/Толстой.txt", encoding="utf8")
for i in range(7):
    print(f.readline(), end="")
f.close()
```

```
   -- Eh bien, mon prince. Gênes et Lucques ne sont plus que des apanages,
des поместья, de la famille Buonaparte.  Non, je  vous préviens, que si vous
ne  me dites pas, que nous avons la guerre, si vous vous permettez encore de
pallier  toutes les infamies, toutes les  atrocités  de cet  Antichrist  (ma
parole, j'y  crois) -- je  ne  vous  connais plus, vous n'êtes plus mon ami,
vous n'êtes  plus  мой  верный  раб,  comme  vous  dites.  [1]  Ну,
здравствуйте, здравствуйте.  Je vois  que  je  vous fais  peur, [2]
```

Также можно считать текстовый файл в список строк с помощью метода `readlines`:

```
f = open("files/Толстой.txt", encoding="utf8")
lines = f.readlines()
print('Type: %s, length: %d' % (type(lines), len(lines)))
print(lines[10])
f.close()
```

```
Type: <class 'list'>, length: 12128
    князя  Василия,  первого  приехавшего  на  её вечер. Анна  Павловна  кашляла
```

Файл может быть построчным итератором (выведем только 10 строк):

```
f = open("files/Толстой.txt", encoding="utf8")
for number, line in enumerate(f):
    print(line)
    if number > 8:
        break
f.close()
```

```
    -- Eh bien, mon prince. Gênes et Lucques ne sont plus que des apanages,
```

```
des поместья, de la famille Buonaparte.  Non, je  vous préviens, que si vous

ne  me dites pas, que nous avons la guerre, si vous vous permettez encore de

pallier  toutes les infamies, toutes les  atrocités  de cet  Antichrist  (ma

parole, j'y  crois) -- je  ne  vous  connais plus, vous n'êtes plus mon ami,

vous n'êtes  plus  мой  верный  раб,  comme  vous  dites.  [1]  Ну,

здравствуйте, здравствуйте.  Je vois  que  je  vous fais  peur, [2]

садитесь и рассказывайте.

       Так говорила в июле 1805 года известная Анна Павловна Шерер, фрейлина и

приближенная  императрицы  Марии  Феодоровны,  встречая важного и  чиновного
```

7. Запись в файл

Для записи в файл также есть два режима: `w` (если файл существовал, его содержимое будет потеряно) и `a` — запись идет в конец файла.

После выбора режима можно также ввести и символ "+". Посмотрите в документации по функции `open`, что означает такая конструкция.

Один из способов записать информацию в файл — метод `write`. Если мы хотим сделать запись в середину файла, должны сначала спозиционироваться на место предполагаемой записи (метод `seek`), а уже потом записывать (метод `write`). Метод `write` возвращает количество записанных символов.

```
f = open("files/example.txt", 'w')
print(f.write('123\n456'))
print(f.seek(3))
print(f.write('34352'))
f.close()
f = open("files/example.txt", 'r')
print(f.read())
f.close()
```

```
7
3
5
12334352
```

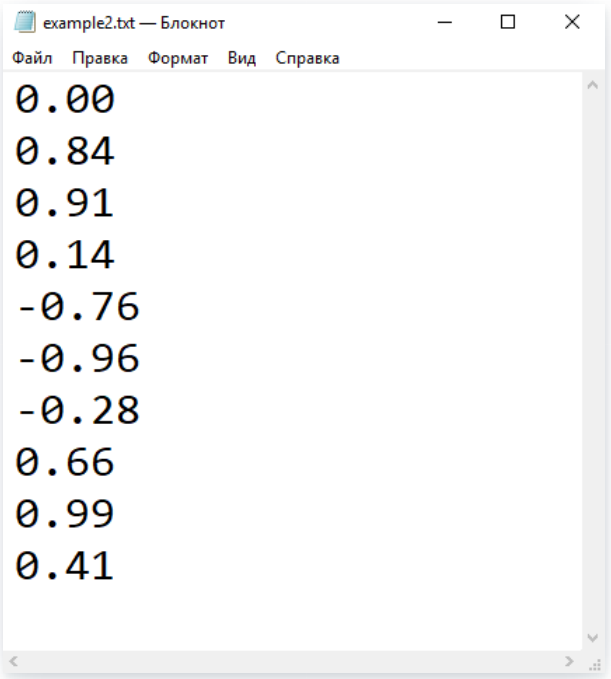
Построчно пройдитесь по приведенной программе, чтобы лучше понять, как она работает. Заметьте, что если мы используем `seek`, то данные при записи все равно будут стерты. Этот процесс похож на рисование фломастером: рисуя поверх, вы закрашиваете старый рисунок.

Второй способ записи в файл — стандартная функция `print`. Для этого применяется именованный параметр `file`.

Например:

```
from math import sin

f = open("files/example2.txt", 'w')
for i in range(10):
    print("%0.2f" % sin(i), file=f)
f.close()
```



Для записи данных в бинарный файл в функцию `write()` надо передать переменную типа `bytes`. Например, так:

```
f = open("files/ex_bin.dt", 'wb')
data = [1, 2, 3, 4, 5]
f.write(bytes(data))
f.close()
```

Обратите внимание: в примере мы преобразовываем список в поток байт, чтобы потом записать его в файл.

8. Закрытие файлов

Операционная система контролирует доступ к файлам. Если какая-то программа открыла файл для записи, все попытки любых других программ изменить содержание файла заблокируются для сохранения целостности.

Поэтому после того как работа с файлом закончена, файл надо **отпустить**, закрыв его методом `close`, что мы делали практически во всех примерах.

```
f.close()
```

Повторим, что после завершения программы все файлы, которая она использовала в своей работе, автоматически закроются. Но **хороший стиль программирования** — это как можно раньше закрыть файл.

Поэтому всегда следуйте принципу: «Не нужен файл — отпусти его!»

И самое последнее: для того чтобы файл закрывался автоматически даже в случае ошибок во время выполнения других операций, в языке Python есть блок `with`. Его назначение шире, но в нашем случае он дает закрыть файл после выхода из блока.

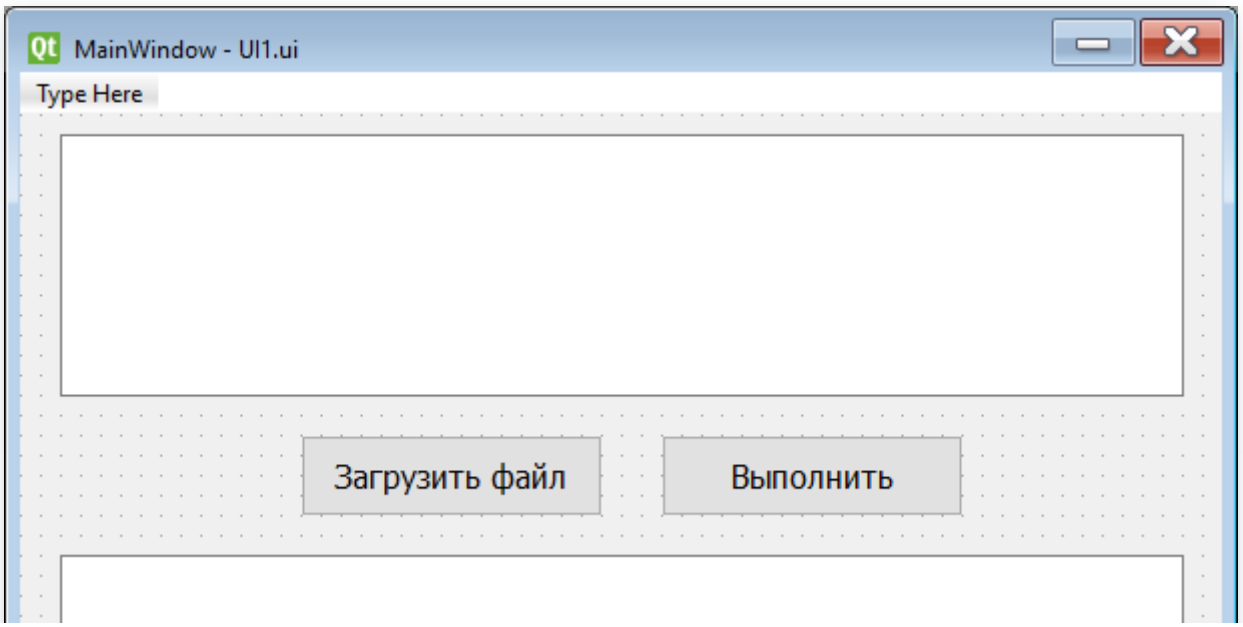
Его синтаксис такой:

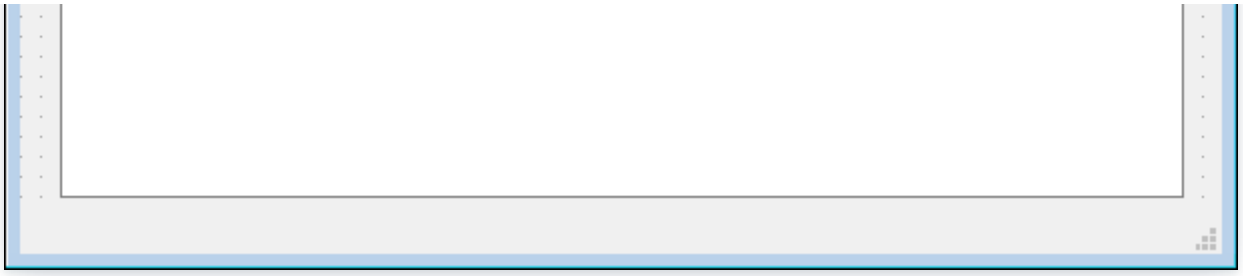
```
with open('files/Толстой.txt', 'rt') as f:
    read_data = f.read()
print(read_data[:100])
print(f.closed)
```

```
        -- Eh bien, mon prince. Gênes et Lucques ne sont plus que des apanages,
    des
True
```

9. PyQT

Одним из самых широко используемых виджетов в PyQT для отображения текстовой информации является **QTextBrowser**. Его возможности очень велики. Начнем с того, что создадим интерфейс приложения через QtDesigner и посмотрим на те свойства этого виджета, которые можно ему задать.

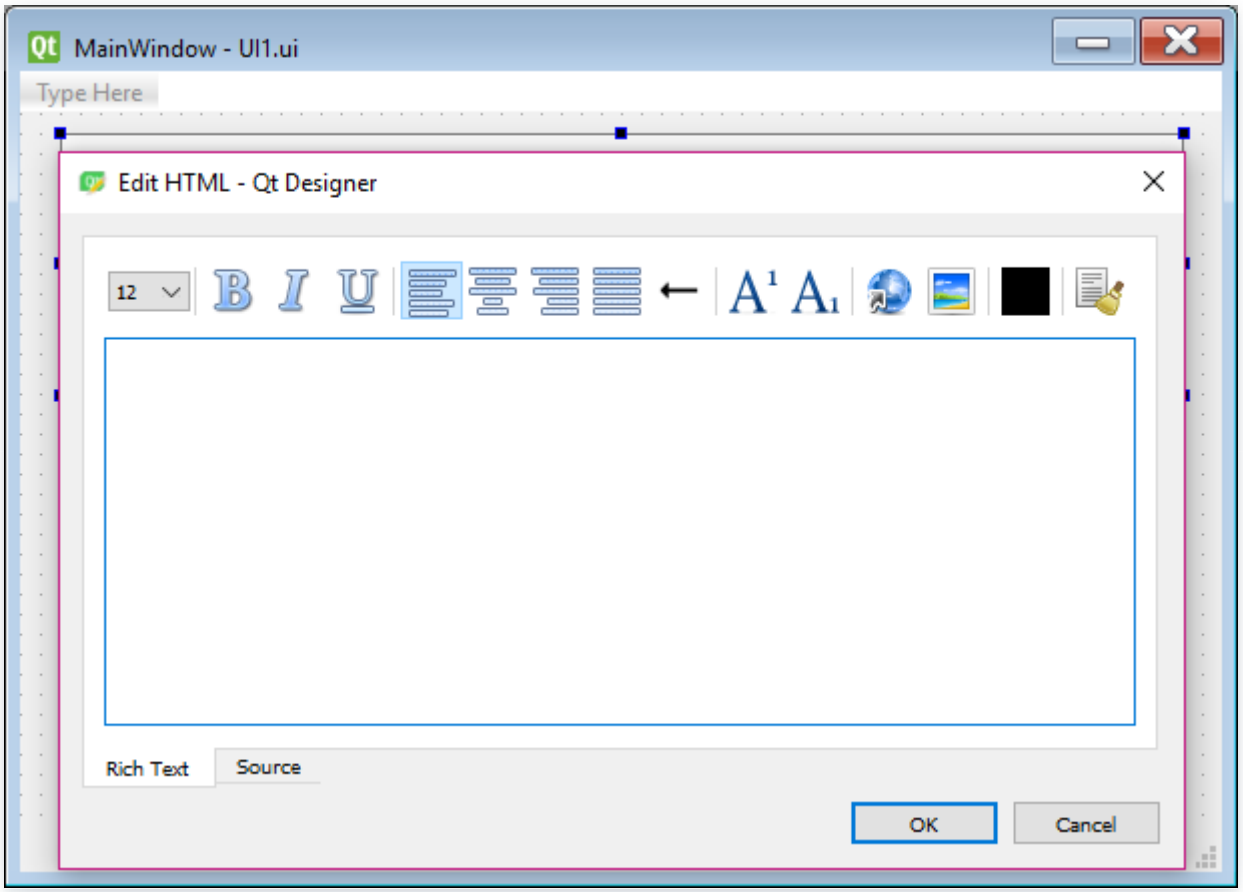




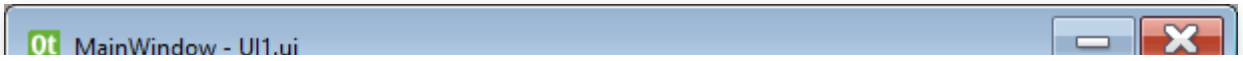
Кроме стандартных настроек самого виджета (размер виджета, его расположение и т. д.), есть группа настроек, отвечающих за шрифт.

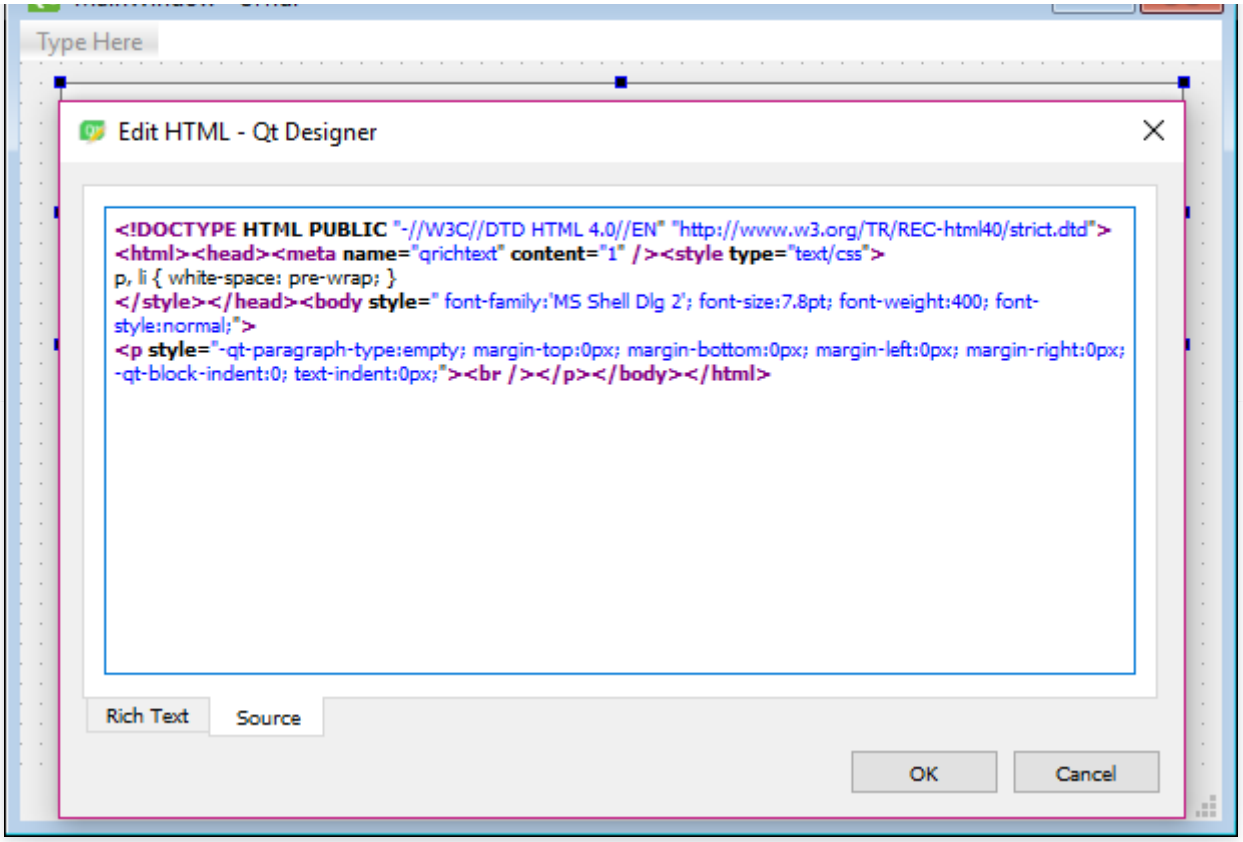
font	A [MS Shell Dlg 2, 8]
Family	MS Shell Dlg 2
Point Size	8
Bold	☐
Italic	☐
Underline	☐
Strikeout	☐
Kerning	☑
Antialiasing	PreferDefault

Интересно, что для того чтобы отображать различный текст, используется язык разметки HTML, с которым вы должны были познакомиться летом. Так что можно вставить свой HTML-код для красивого оформления текста. Для этого в редакторе свойств нужно выбрать поле html, после чего откроется редактор, в котором можно верстать различные тексты. Как в привычном Word-подобном интерфейсе,



так и используя просто HTML.





Этот виджет может использоваться не только для отображения информации, но и для ее ввода. Но, поскольку в данном уроке мы будем получать информацию из файлов, мы можем смело настроить режим **readOnly**. Это тоже можно сделать в редакторе свойств QtDesigner.

Теперь реализуем нашу первую программу. Интерфейс можно собрать самостоятельно в дизайнере (или создать в коде), а можно взять вот **тут**. Она будет получать какую-то информацию из файла и выводить ее в обратном порядке.

Код вашего класса должен выглядеть примерно вот так:

```
class TextBrowserSample(QMainWindow):
    def __init__(self):
        super(TextBrowserSample, self).__init__()
        uic.loadUi('TextBrowserSample.ui', self)
        self.loadButton.clicked.connect(self.load_file)
        self.processButton.clicked.connect(self.process_data)

    def load_file(self):
        pass

    def process_data(self):
        pass
```

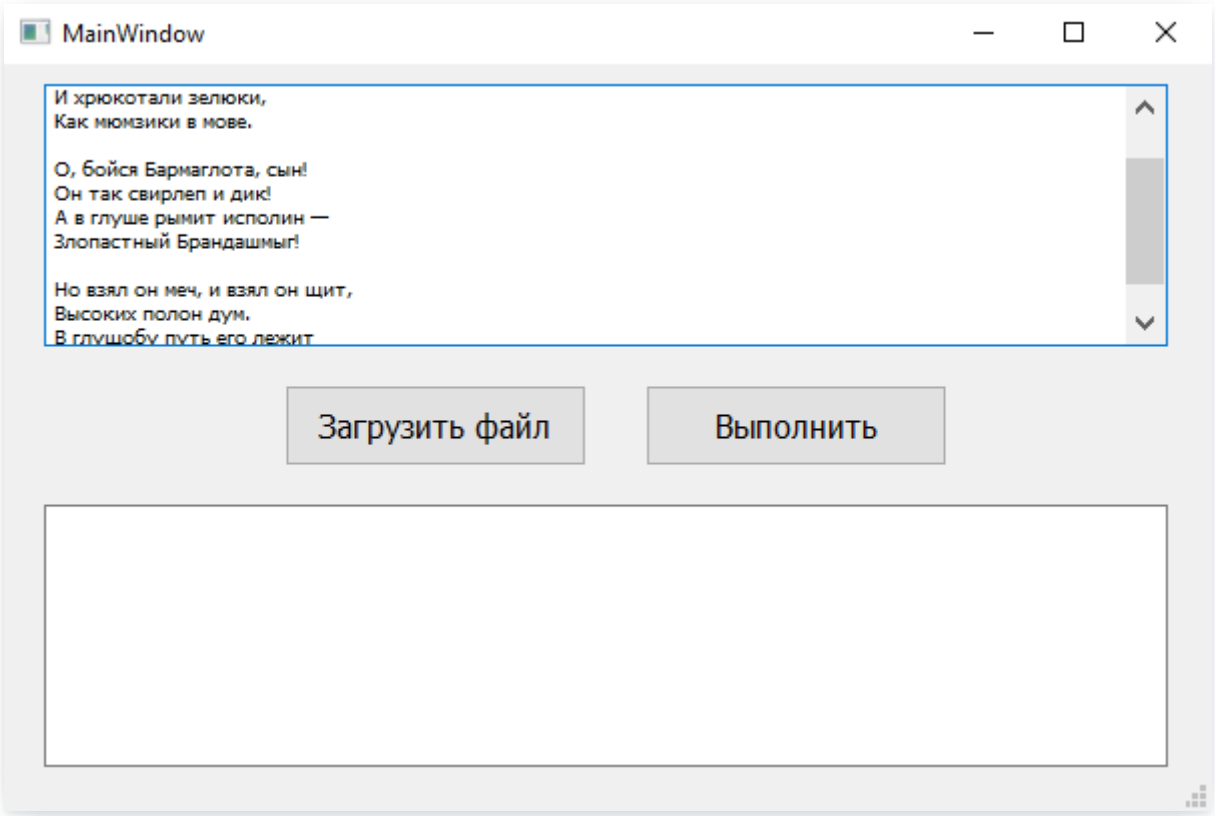
Для начала необходимо загрузить содержимое текста в наш виджет. Напишем для этого простейшую функцию:

```
def load_file(self):
    try:
        with open('input.txt', 'r',
```

```
                encoding='utf8') as f:
            # Если файла нет, то мы получим исключение FileNotFoundError
            self.inputText.setText(f.read())
    except FileNotFoundError as ex:
        self.statusBar().showMessage('В директории нет файла input.txt')
```

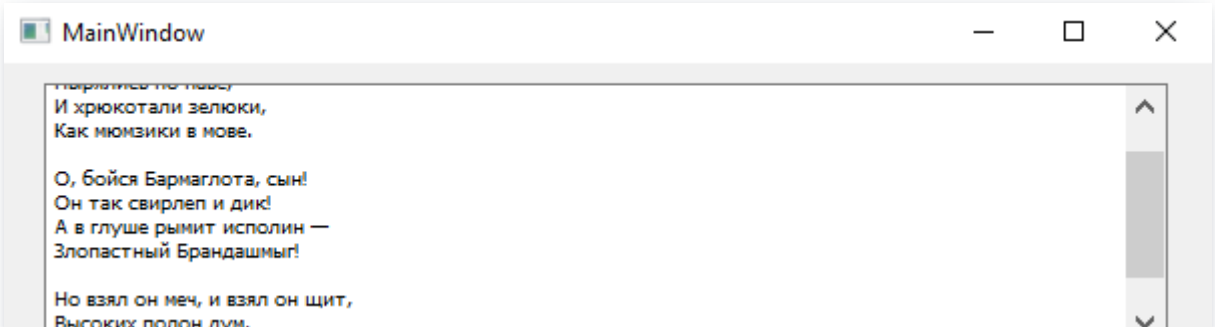
Метод `setText` позволяет загрузить переданный текст в виджет. Интересно, что, если размер текста превышает размеры виджета по вертикали или горизонтали, автоматически появляется слайдер, позволяющий прокручивать текст.

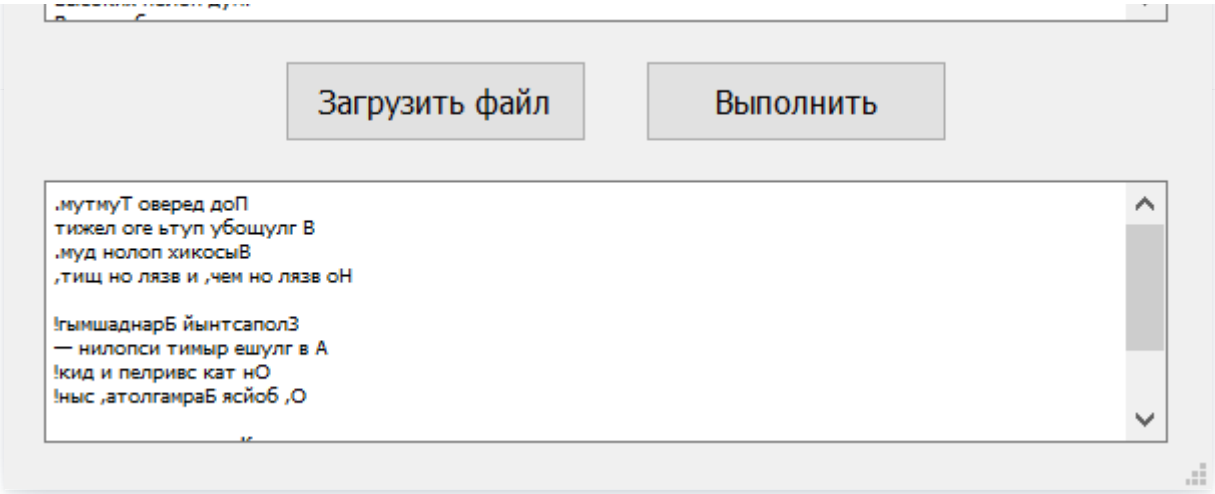
Дополнительно давайте обработаем случай, когда файла input.txt нет в директории с нашей программой: тогда при попытке получить доступ к файлу будет выброшено исключение FileNotFoundError. При получении такого исключения будем выводить в StatusBar соответствующее сообщение.



Теперь мы можем работать с содержимым виджета. Если мы хотим работать с ним как с обычным текстом, игнорируя HTML-теги и форматирование, можно воспользоваться методом `toPlainText()`.

```
def process_data(self):
    data = self.inputText.toPlainText()
    self.outputText.setText(data[::-1])
```





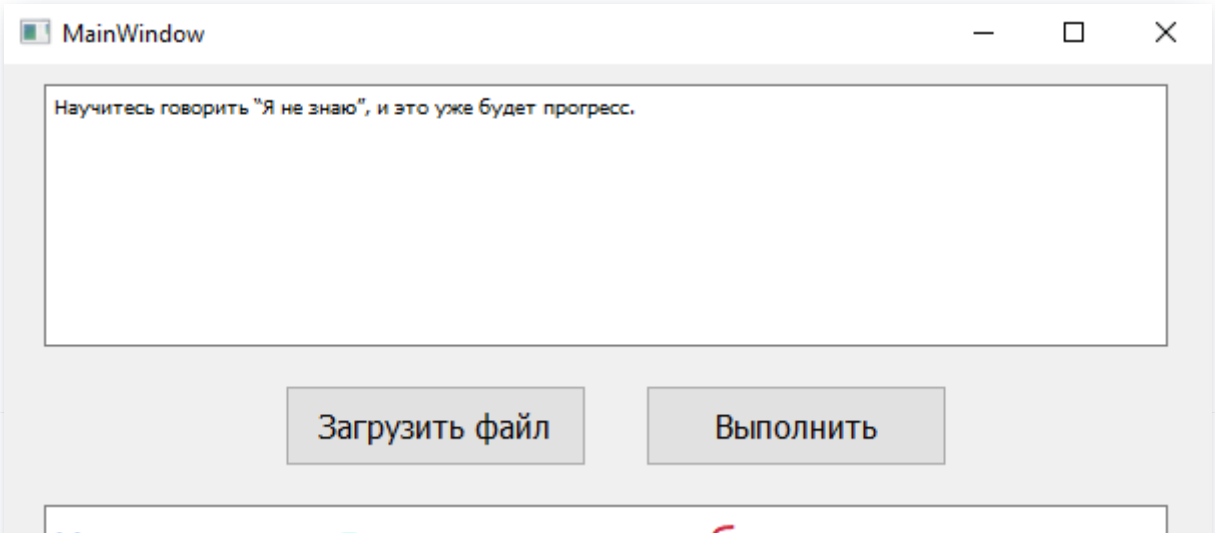
Теперь давайте вспомним, что мы можем использовать различное оформление текста. Напишем программу, которая будет окрашивать каждую букву, полученную из файла, в случайный цвет и задавать ей случайный размер. Для этого мы будем использовать HTML-тег `<font color="цвет">...</font>`, а цвет — это случайное шестнадцатиричное число от 0 до 0xFFFFFF.

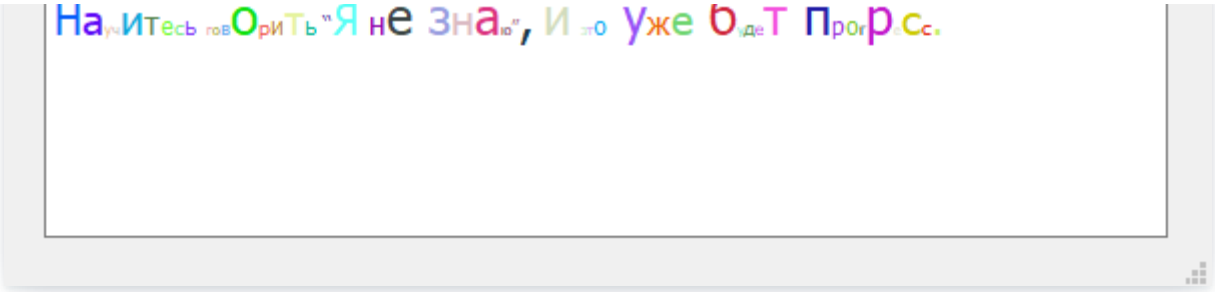
Напишем другую функцию-слот для обработки нажатия на кнопку «Выполнить»:

```
def color_text(self):
    data = self.inputText.toPlainText()
    HTML = ""
    for i in data:
        color = "#{:06x}".format(random.randrange(0, 0xFFFFFF))
        HTML += "<font color='{}' size = {} >{}</font>".format(
            color, random.randrange(1, 8), i)
    self.outputText.setHtml(HTML)
```

Как он работает? Сначала, как и в первом примере, содержимое текста заносится в переменную. Затем, используя модуль `random` (не забудьте его импортировать!), мы получаем случайное шестнадцатиричное число — наш цвет, а затем добавляем наш символ в «обрамлении» тега, отвечающего за цвет шрифта и его размер. Подробнее об HTML-тегах можно прочитать в Интернете, например, **здесь**.

Ну и для того, чтобы занести форматированный с помощью HTML текст в виджет, понадобится метод `setHtml()`.





В этом уроке мы работали с только с простыми текстовыми файлами, однако форматов гораздо больше. В следующих уроках мы поработаем и с другими популярными форматами.

10. Важная информация о текстовых файлах.

Согласно **определению из стандарта POSIX**, который тоже пришёл к нам из эпохи печатных машинок:

Строка

Строка — это последовательность из нуля или более символов, не являющихся символом новой строки, и последнего символа новой строки.

Почему важен этот стандарт? Возможно большое количество способов реализовать одно и то же, и только благодаря стандартам, таким как POSIX, мы имеем сейчас огромное количество качественного программного обеспечения, которое не конфликтует друг с другом.
Т.е. если вы не ставите символ переноса строки в конце строки, то формально по стандарту такая строка не является валидной (правильной). Множество программ, которыми пользуются каждый день, написано в согласии с этим стандартом, и они просто не могут правильно обрабатывать такие "сломанные" строки.
Давайте, например, через Python создадим такой файл со сломанными строками:

```
with open("broken.txt", "w") as f:
    f.write("qwe\n")
    f.write("asd\n")
    f.write("zxc")
```

Сколько по-вашему в этом файле строк? Три? Давайте посмотрим, что об этом файле думает утилита **wc**, которая с флагом **-l** умеет считать количество строк в файле (утилита работает только в Linux/MacOS или gitBash):

```
$ wc -l broken.txt
2 broken.txt
```

Найдено только две строки!
Давайте создадим еще один файл:

```
with open("broken2.txt", "w") as f:
    f.write("rty\n")
```

```
    f.write("fgh\n")
    f.write("vbn")
```

И попробуем теперь склеить два созданных файла при помощи утилиты **cat**:

```
$ cat broken.txt broken2.txt
qwe
asd
zxcrty
fgh
vbn
```

И опять какой-то странный результат! В большинстве случаев это не то, чего вы бы ожидали, но вполне возможны ситуации, когда вам нужен именно такой результат. Именно поэтому утилита cat не может самостоятельно вставлять отсутствующие символы переноса строки, иначе это сделало бы её поведение неодинаковым.

Возможно, такая маленькая деталь, как перенос строки в конце файла и не кажется очень важной, а тема вообще кажется спорной, но у нас нет другого выбора, кроме как принять это правило за данность и просто выработать привычку всегда ставить символ новой строки в любых текстовых файлах, даже если этого не требуется явно. Это считается распространённой хорошей практикой, и как минимум убережёт вас и ваших коллег от всяких неожиданных эффектов.

В текстовом редакторе это выглядит как лишняя пустая строка в конце файла.