ChatGPT

# AnimatedButton (PyQt5)

## Назначение виджета

**AnimatedButton** – это пользовательский виджет на основе PyQt5, представляющий собой анимированную кнопку, которая по требованию раскрывается в виджет типа `QSpinBox` (числовой счётчик). В компактном (свёрнутом) состоянии виджет выглядит и ведёт себя как обычная кнопка. При активации (например, при нажатии) он плавно расширяется, открывая встроенный счётчик `QSpinBox`, позволяющий пользователю выбрать числовое значение. После выбора значения виджет можно свернуть обратно до состояния кнопки. Такая концепция экономит место в интерфейсе и улучшает UX, показывая полный элемент управления (spin box) только тогда, когда это необходимо пользователю.

AnimatedButton полезен в случаях, когда требуется ввод числового значения, но постоянно отображать `QSpinBox` на экране нежелательно. Например, это может быть выбор количества единиц товара: изначально отображается компактная кнопка (с текущим значением или иконкой), при нажатии – раскрывается поле для ввода числа с кнопками увеличения/уменьшения.

## Машина состояний `ButtonState`

Для управления поведением раскрывающейся кнопки AnimatedButton использует внутреннюю машину состояний, определяемую перечислением `ButtonState`. Это обеспечивает корректное выполнение анимации и реагирование на действия пользователя. Состояния `ButtonState` включают в себя следующие этапы работы виджета:

- **Collapsed (Свёрнуто)** – исходное состояние. `QSpinBox` скрыт, виден только сам элемент кнопки. В этом состоянии виджет занимает минимальный размер.
- **Expanding (Раскрытие)** – промежуточное состояние анимации раскрытия. Возникает при начале процесса расширения кнопки. В этом состоянии происходит плавное изменение размеров (например, ширины) виджета, и `QSpinBox` постепенно становится видимым.
- **Expanded (Развёрнуто)** – состояние полностью раскрытого виджета. `QSpinBox` полностью отображается и доступен для ввода/изменения значения. Пользователь может изменить значение с помощью стрелок или ввода с клавиатуры, как в обычном `QSpinBox`.
- **Collapsing (Сворачивание)** – промежуточное состояние анимации обратного схлопывания. Наступает, когда виджет инициирует закрытие `QSpinBox` и возвращение к компактному виду кнопки. Происходит плавное уменьшение размеров до исходных.

Машина состояний гарантирует, что переходы между этими состояниями осуществляются последовательно и корректно. Так, например, повторное нажатие кнопки во время состояния `Expanding` или `Collapsing` не будет обработано мгновенно (во избежание конфликтов анимации) – виджет либо дождётся окончания текущей анимации, либо игнорирует лишние события до завершения перехода. После достижения конечного состояния (`Expanded` или `Collapsed`) виджет готов реагировать на новые действия пользователя.

Внутренняя реализация может основываться на Qt State Machine (`QStateMachine`) или аналогичной логике: определены состояния и переходы (triggered, например, сигналами о клике или потере фокуса). При переходах используются анимации свойств (например, изменение ширины) для плавного раскрытия/сворачивания. Таким образом, AnimatedButton сочетает в себе логику кнопки и спинбокса, сохраняя интуитивно понятный цикл состояний: **Свёрнуто → (нажатие) → Раскрытие → Развёрнуто → (завершение ввода/вторичное действие) → Сворачивание → Свёрнуто**.

## Публичные методы

AnimatedButton предоставляет следующий интерфейс (публичные методы) для настройки и управления виджетом:

`__init__(parent=None, ...)`

Конструктор виджета. Инициализирует экземпляр AnimatedButton, создавая внутренние компоненты и устанавливая начальное состояние. В конструкторе выполняется:

- Создание базового представления кнопки (в свёрнутом виде). Этот элемент отображает либо текст/значение, либо иконку – в зависимости от дизайна – и реагирует на нажатия как обычная кнопка.
- Создание вложенного элемента `QSpinBox`, который будет показан при раскрытии. Spin box настраивается со значениями по умолчанию (например, диапазон 0–99, шаг 1, начальное значение 0) и изначально скрыт или сжат.
- Настройка компоновки: обычно кнопка и spin box располагаются рядом в одном горизонтальном Layout, при этом начальная ширина spin box сведена к нулю или он скрыт, чтобы виджет выглядел как кнопка. Также может задаваться отступ (padding) вокруг содержимого и промежуток (spacing) между кнопкой и полем ввода согласно теме оформления.
- Инициализация темы по умолчанию через `_ThemeConfig.create()`. Применяются стандартные значения оформления (цвета текста, фона, скорость анимации и т.д.) если не указано иного.
- Установка начального состояния `ButtonState` в **Collapsed** (свёрнуто).
- Подключение внутренних сигналов: например, сигнал `QSpinBox.valueChanged` привязывается к собственному сигналу `valueChanged` виджета, а событие нажатия на кнопку инициирует переход к состоянию **Expanding** (через вызов `expand_button()`) и испускает сигнал `clicked`.

Параметр `parent` (типично `QWidget` или окно) указывается по желанию для встраивания AnimatedButton в существующий интерфейс. Дополнительные параметры могут отсутствовать, т.к. настройка значений и диапазона выполняется отдельными методами (например, `setValue`) или непосредственным доступом к spin box (см. раздел *Примеры использования*). После создания экземпляра AnimatedButton его можно добавлять в макет (`QLayout`) аналогично любому другому виджету.

### `set_scale(scale: float)`

Устанавливает масштаб виджета. Этот метод упрощает изменение общего размера и пропорций AnimatedButton – включая размеры текста, отступов и пр. – одним числовым коэффициентом.

- Параметр `scale` — коэффициент масштабирования (например, `1.0` – стандартный размер по умолчанию, `1.5` – на 50% больше, `0.8` – уменьшенный на 20% и т.д.).
- Вызов `set_scale` применяет новый масштаб к текущей теме оформления виджета. Внутренне могут пересчитываться некоторые параметры стиля: размер шрифта (`text_scale`), поля, отступы, размеры стрелок `QSpinBox` и другие, чтобы виджет пропорционально увеличился или уменьшился.
- Например, если `scale = 1.2`, шрифт и элементы интерфейса станут несколько крупнее, а если `0.5` – виджет уменьшится (что может быть полезно для компактного размещения или в составе других компонентов).

Метод удобен для быстрой адаптации компонента под разные DPI экраны или под разные контексты использования (обычный режим vs. режим с большими элементами для сенсорного ввода). Важно отметить, что `set_scale` следует вызывать **до** отображения виджета или сразу после создания, чтобы перерасчёт размеров произошёл корректно. Повторный вызов при необходимости изменит масштаб на лету (виджет обновит оформление).

### `apply_custom_style(overrides: dict)`

Применяет кастомизированный стиль к виджету, используя словарь переопределений. Этот метод позволяет разработчику настроить внешний вид AnimatedButton под стилистику своего приложения, изменяя такие параметры как цвета, отступы, скорость анимации и прочее.

- Параметр `overrides` – словарь, в котором ключами являются имена настраиваемых свойств, а значениями – новые значения этих свойств. Поддерживаемые ключи и их значения подробно описаны в разделе внутреннего класса _ThemeConfig. Примеры ключей: `'padding'`, `'spacing'`, `'text_color'`, `'active_color'`, `'anim_speed'`, `'text_scale'` и др.
- При вызове `apply_custom_style` виджет берёт базовую тему по умолчанию, переопределяет указанные свойства на значения из словаря и применяет их. Например, если передать `{'text_color': 'white', 'active_color': '#008000', 'padding': 8}`, виджет обновит цвет текста на белый, цвет активного состояния на зелёный (в данном случае заданный кодом `#008000`), а отступы вокруг содержимого установит равными 8px.
- Метод автоматически перерисовывает виджет с учётом новых стилей. Цвета преобразуются во внутренние объекты `QColor` (можно передавать как объекты `QtGui.QColor`, так и строковые названия/HEX-коды). Числовые параметры (отступы, размеры) принимаются в пикселях, а скорость анимации – в миллисекундах.
- `apply_custom_style` можно вызывать как до отображения виджета, так и динамически в процессе работы приложения, чтобы сменить тему "на лету". Например, можно реализовать переключение светлой/тёмной темы, меняя цвета через этот метод.

Обратите внимание: если ранее вызывался `set_scale`, его эффект может быть частично переопределён новым стилем (особенно для `text_scale` или размеров). Обычно рекомендуется либо сначала задать масштаб `set_scale`, а затем применять дополнительные стилевые правки через `apply_custom_style`, либо указывать желаемый масштаб шрифта прямо в словаре (ключ `'text_scale'`). Внутренне `apply_custom_style` также может

использовать `_ThemeConfig.create()` для слияния пользовательских параметров с дефолтными.

### `value() -> int`

Возвращает текущее числовое значение, установленное в spin box виджета. Этот метод аналогичен `QSpinBox.value()`. Его можно вызывать в любой момент, чтобы получить выбранное пользователем значение.

- Если AnimatedButton в свёрнутом состоянии, `value()` всё равно вернёт последнее установленное значение (даже если пользователь ещё не раскрывал виджет). По умолчанию, сразу после создания, это может быть 0 (или иное значение по умолчанию, если было изменено).
- В развернутом состоянии метод отражает текущее значение, которое пользователь мог изменить прямо перед вызовом.
- Возвращаемый тип – `int`. Если требуется значение другого типа (например, `float`), компонент AnimatedButton не предоставляет прямого метода, но можно использовать `QDoubleSpinBox` аналогом (в данной реализации рассматривается целочисленный spinbox).

Пример использования:

```
current = anim_button.value()
print(f"Selected value: {current}")
```

Метод удобен для единоразового опроса значения. Для отслеживания изменений в реальном времени см. сигнал `valueChanged`.

### `setValue(val: int)`

Устанавливает значение spin box на указанное число `val`. По сути, программно изменяет текущее значение так же, как если бы пользователь ввёл его вручную, с последующей эмиссией сигнала `valueChanged`.

- `val` – целое число, которое нужно установить. Если `val` выходит за пределы текущего диапазона `QSpinBox`, значение будет откорректировано до ближней границы (например, если `val` больше максимума, установится максимум).
- При вызове `setValue` внутренний `QSpinBox` обновляет отображаемое число. Если AnimatedButton сейчас в состоянии **Collapsed**, обычно на кнопке отображается последнее значение, поэтому после `setValue` при свёрнутом состоянии виджет тоже должен обновить текст/метку на кнопке, чтобы отразить новое значение.
- `setValue` **генерирует сигнал** `valueChanged(int)` так же, как при ручном изменении пользователем. Это позволяет унифицировать обработку – внешний код, подписанный на `valueChanged`, получит уведомление об изменении вне зависимости от того, было оно инициировано пользователем или программно.
- Метод можно использовать для начальной инициализации значения сразу после создания виджета (если не устраивает дефолт 0) или для сброса/изменения значения по ходу работы программы.

Пример: `anim_button.setValue(10)` – установит значение 10 и обновит интерфейс виджета.

### `expand_button()`

Программно раскрывает кнопку, начиная анимацию перехода к состоянию **Expanded**. Этот метод выполняет то же действие, что и нажатие пользователем – инициирует разворачивание виджета.

- Если AnimatedButton уже развёрнут или находится в процессе раскрытия, вызов `expand_button()` не приведёт к дублированию анимации (виджет либо уже раскрыт, либо скоро будет раскрыт). В таких случаях метод может игнорироваться или срабатывать условно. Обычно имеет смысл вызывать его, когда виджет в **Collapsed** состоянии.
- При вызове метод переключает внутреннее состояние на **Expanding**, делает видимым компонент `QSpinBox` (если он был скрыт) и запускает анимацию изменения размера. Чаще всего анимация затрагивает ширину виджета или другой свойство, обеспечивая плавный визуальный эффект.
- По завершении анимации внутреннее состояние станет **Expanded**. Если предусмотрены какие-либо действия по завершении (например, автофокус на поле ввода), `expand_button` их выполнит: часто при полном раскрытии фокус устанавливается в `QSpinBox` для удобства ввода с клавиатуры.
- Обычно необходимость ручного вызова `expand_button` ограничена случаями, когда разработчик хочет раскрыть компонент без прямого пользовательского клика (например, в ответ на какой-то другой сигнал/событие в интерфейсе). В большинстве случаев пользователи просто нажимают на сам AnimatedButton, и он раскрывается автоматически.

### `collapse_button()`

Обратное действие к предыдущему методу – программно сворачивает виджет обратно в состояние кнопки (инициирует переход к **Collapsed**).

- Если виджет уже свёрнут или находится в процессе сворачивания, повторный вызов `collapse_button()` не имеет эффекта (будет проигнорирован).
- При вызове, если AnimatedButton в состоянии **Expanded**, метод запускает анимацию схлопывания. `QSpinBox` может терять фокус (если он имелся) и постепенно скрываться, обычно посредством уменьшения ширины/размера виджета до компактного. Внутреннее состояние переключается на **Collapsing** на время анимации.
- После завершения анимации виджет возвращается в состояние **Collapsed** – `QSpinBox` полностью скрыт. Если на кнопке должно отображаться значение, оно обновлено и видно.
- `collapse_button` полезен, например, чтобы автоматически закрыть ввод после выбора значения. Разработчик может вызывать его, например, в обработчике `valueChanged`, чтобы свернуть компонент сразу после изменения значения (если такой сценарий UX необходим).

Оба метода `expand_button()` и `collapse_button()` работают асинхронно (запуская анимацию). Если нужно узнать о факте завершения раскрытия/схлопывания, это может достигаться отслеживанием изменения состояний (например, через логи или дополнительными сигналами, которых по умолчанию нет, но можно косвенно определить момент по наступлению `Expanded` / `Collapsed` состояний, например, ловя фокус).

## Сигналы

AnimatedButton определяет два ключевых сигнала для взаимодействия с внешним кодом (помимо стандартных Qt-сигналов внутренних компонентов):

- `valueChanged(int newValue)` – испускается при изменении значения счётчика. Эквивалентен сигналу `QSpinBox.valueChanged<int>` и срабатывает каждый раз, когда пользователь выбирает новое число или когда программа вызывает `setValue`. Обработчик, подключенный к этому сигналу, получает новое значение `newValue` типа `int`. Используется для реагирования на ввод пользователя – например, обновления связанных данных или интерфейса в соответствии с выбранным числом.
- `clicked()` – сигнал клика по **кнопке** AnimatedButton. Он эмитится в момент нажатия на сам виджет (на его кнопку) до начала анимации раскрытия. Таким образом, этот сигнал уведомляет, что пользователь инициировал раскрытие (или нажал на компактную кнопку). Если AnimatedButton уже был в раскрытом состоянии и пользователь вновь нажимает на его кнопку (возможно, чтобы закрыть), `clicked()` также сработает. По семантике этот сигнал аналогичен `QPushButton.clicked` – он не передаёт дополнительных параметров, просто факт нажатия.

  **Примечание:** В обычном сценарии нет необходимости вручную вызывать `expand_button()` в обработчике `clicked()` – компонент и так настроен раскрывать себя при клике. Однако сигнал `clicked` может быть полезен, чтобы выполнить какие-то сопутствующие действия в момент нажатия (например, изменить другой элемент интерфейса, проиграть звук и т.д.).

Сигналы `valueChanged` и `clicked` можно подключать стандартным образом:

```
anim_button.valueChanged.connect(handle_value)
anim_button.clicked.connect(handle_click)
```

где `handle_value` и `handle_click` – ваши слоты/функции.

Помимо этих, AnimatedButton может наследовать другие сигналы от базовых классов (если он, например, унаследован от `QWidget` или `QAbstractButton`). Однако, пользоваться ими напрямую обычно нет необходимости – главное взаимодействие происходит через указанные выше сигналы и методы.

## Внутренний класс `_ThemeConfig`

Внутри AnimatedButton определён класс `_ThemeConfig`, отвечающий за хранение и создание конфигурации темы оформления виджета. Этот класс, хотя и является внутренним (начинается с подчёркивания, не предназначен для прямого использования вне модуля), играет важную роль при кастомизации внешнего вида.

Метод `_ThemeConfig.create(overrides: dict) -> dict` принимает словарь переопределений стилей и возвращает готовую конфигурацию (чаще всего тоже в виде словаря) с установленными всеми необходимыми ключами. Работает это следующим образом:

1. В `_ThemeConfig` зашиты **значения по умолчанию** для всех параметров оформления AnimatedButton. Пример набора параметров по умолчанию:
2. `'padding'`: внутренний отступ вокруг содержимого (например, 4 пикселя по умолчанию).
3. `'spacing'`: расстояние между элементами (кнопкой и полем ввода) в раскрытом виде (например, 2 пикселя).
4. `'text_color'`: цвет текста на кнопке и в spinbox (например, чёрный или белый в зависимости от темы).
5. `'active_color'`: акцентный цвет для выделения активного состояния (например, синий или зелёный по умолчанию – используется для рамки или фона кнопки при фокусе/нажатии).
6. `'anim_speed'`: длительность анимации в миллисекундах (например, 300 мс по умолчанию, что даёт быструю, но заметную анимацию).
7. `'text_scale'`: масштаб шрифта (1.0 по умолчанию – базовый размер шрифта, используемый системой или заданный явно; может применяться для увеличения/уменьшения текста относительно стандартного шрифта приложения).
8. **Другие**: могут быть и дополнительные параметры, определяющие цвета и размеры разных аспектов виджета. Например, это могут быть `'background_color'` (цвет фона кнопки в обычном состоянии), `'hover_color'` (цвет при наведении курсора), `'border_color'` (цвет рамки виджета, если есть рамка), `'border_radius'` (радиус скругления углов, если виджет рисует скруглённые края), `'arrow_color'` (цвет стрелок `QSpinBox`), `'arrow_size'` (размер стрелочек) и т.п. – полный список зависит от реализации. В документации и коде AnimatedButton обычно явно указано, какие ключи поддерживаются.
9. При вызове `_ThemeConfig.create(overrides)`, сначала берётся шаблон словаря с дефолтными значениями. Затем функция пробегается по словарю `overrides` и заменяет соответствующие ключи на новые значения. Любые ключи, не указанные в `overrides`, останутся со значениями по умолчанию.
10. Результатом `create` является **итоговый словарь конфигурации**, который содержит **все** необходимые ключи для оформления. Этот словарь затем используется AnimatedButton для применения стилей (например, цвета текста применяются через `QPalette` или стили, отступы и размеры – через методы Qt или StyleSheet, скорость анимации – для настройки `QPropertyAnimation` и т.д.).

Назначение `_ThemeConfig` – инкапсулировать знание о параметрах темы в одном месте. Пользователю компонента **не требуется** вызывать `_ThemeConfig.create` напрямую. Вместо этого, публичный метод `apply_custom_style` служит оболочкой, внутри которой `_ThemeConfig` будет применён. Однако, зная о структуре `_ThemeConfig`, разработчик понимает, **какие именно параметры можно переопределить** в словаре стилей: - `padding` – отступы внутри виджета (толщина полей вокруг текста/числа). - `spacing` – промежуток между кнопкой и полем ввода при раскрытии. - `text_color` – цвет текста (на кнопке и, возможно, содержимого `QSpinBox`). - `active_color` – цвет акцента для активного элемента (например, фон или рамка кнопки при нажатии/фокусе, цвет стрелок при наведении и т.д.). - `anim_speed` – продолжительность анимации (мс). - `text_scale` – относительный размер шрифта. - *Дополнительные параметры*, если поддерживаются: `background_color`, `hover_color`, `border_color`, `border_radius`, и т.д., позволяющие ещё глубже настроить виджет под стиль приложения.

Используя эти ключи, можно тонко настроить внешний вид AnimatedButton без правки кода компонента, а лишь передав нужные значения через `apply_custom_style`. `_ThemeConfig` выступает прослойкой, обеспечивающей, что даже при неполном наборе переопределений остальные параметры будут адекватно заполнены дефолтами.

## Кастомизация стиля

AnimatedButton поддерживает гибкую кастомизацию внешнего вида и размеров, что осуществляется двумя основными способами: методом `apply_custom_style` и методом `set_scale`. Оба могут использоваться независимо или совместно.

**Через `apply_custom_style`:** Этот метод позволяет адресно изменить отдельные аспекты оформления. Например, чтобы подстроить виджет под тёмную тему, можно изменить цвет текста и фона; чтобы ускорить анимацию – уменьшить `anim_speed`; чтобы увеличить отступы – задать большее значение `padding`. Ключи, которые поддерживаются для переопределения, описаны в разделе _ThemeConfig. Важные замечания при кастомизации:

- Цветовые параметры (`text_color`, `active_color`, `background_color` и пр.) можно передавать как строку (имя цвета или HEX-код) или как экземпляр `QColor`. Внутри компонента они будут приведены к QColor и применены через Palette или StyleSheet. Например:

```
anim_button.apply_custom_style({
    'text_color': '#ffffff',       # белый текст
    'active_color': Qt.red,        # красный акцент при фокусе/нажатии
    'background_color': '#333333'  # тёмно-серый фон кнопки
})
```

- Числовые параметры (`padding`, `spacing`, `anim_speed`, размеры) передаются как int. Они интерпретируются в пикселях (для геометрических отступов/размеров) или миллисекундах (для времени анимации). Пример:

```
anim_button.apply_custom_style({
    'padding': 10,     # увеличить внутренний отступ до 10px
    'spacing': 5,      # промежуток между кнопкой и полем 5px
    'anim_speed': 150  # ускорить анимацию до 150 мс
})
```

- Масштаб текста (`text_scale`) – float, обычно >0. Значение 1.0 означает стандартный размер шрифта. Например, `text_scale: 1.2` увеличит шрифт на 20%, а 0.8 – уменьшит на 20%. Это повлияет на отображение числа на кнопке и в поле ввода.
- Комбинируя эти параметры, можно добиться желаемого стиля. **Порядок применения стилей:** если вы используете и `set_scale` и `apply_custom_style`, помните, что последний вызванный метод будет определяющим для пересекающихся настроек.
- Обычно масштаб (`set_scale`) задают первым, чтобы определить общий размер, а затем через `apply_custom_style` донастраивают цвета и мелкие детали.
- Если же вызвать `set_scale` после `apply_custom_style`, то масштабирование может переразмерить шрифт и отступы, но не изменит уже заданные цвета.

**Через** `set_scale`**:** Этот метод, как описано выше, применяется для равномерного изменения размера компонента. По сути, он влияет на несколько стилевых параметров сразу: - Размер шрифта (эквивалентно пропорциональному изменению `text_scale`). - Размеры внутренних отступов `padding` (чем больше масштаб, тем больше отступы, чтобы виджет "рос" равномерно). - Возможно, размеры стрелок spinbox и общий минимум/максимум ширины виджета. - `spacing` между кнопкой и полем тоже может быть увеличен при большом масштабе.

Если вам нужно просто иметь "маленький", "средний" или "большой" вариант AnimatedButton, удобно пользоваться `set_scale`. Например:

```
small_btn = AnimatedButton()
small_btn.set_scale(0.8)  # компактная версия на 20% меньше стандартной

large_btn = AnimatedButton()
large_btn.set_scale(1.5)  # увеличенный вариант, 150% от базового размера
```

После этого при необходимости можно подправить отдельные цвета через `apply_custom_style`, без изменения размеров.

**Цвета и стили по умолчанию:** AnimatedButton изначально пытается вписаться в стиль вашего приложения. Если вы не вызываете `apply_custom_style`, он может использовать палитру по умолчанию (например, фон кнопки – стандартный из темы Qt, цвет текста – из темы и т.п.). Однако, для более гармоничного внешнего вида, рекомендуется задать хотя бы основные цвета, особенно если фон контейнера, где расположен виджет, нестандартный.

Например, на светлом фоне может быть уместно тёмным цветом выделить границы или текст кнопки, а на тёмном – светлым. `active_color` обычно стоит сделать контрастным цветом, показывающим активность (например, фирменный цвет приложения).

Всё оформление происходит программно – использование `apply_custom_style` предпочтительнее применения стилевых таблиц (QSS) напрямую, потому что он обеспечивает согласованность с внутренним состоянием виджета. Тем не менее, AnimatedButton не препятствует применению к нему Qt-стилей: вы можете задать общую тему приложения или применять `setStyleSheet` к AnimatedButton, но тогда нужно понимать, что это может конфликтовать с `_ThemeConfig`. Лучшей практикой будет пользоваться предоставленными методами кастомизации.

## Примеры использования

Ниже приведены примеры того, как AnimatedButton можно использовать в приложении PyQt5. Предполагается, что класс AnimatedButton уже импортирован или определён.

**1. Базовое создание и использование:**

```
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout
# ... импорт AnimatedButton ...

app = QApplication([])
```

```
main_widget = QWidget()
layout = QVBoxLayout(main_widget)

# Создаем AnimatedButton и добавляем в интерфейс
spin_button = AnimatedButton()
layout.addWidget(spin_button)

# Устанавливаем начальное значение (по умолчанию 0, здесь для примера 5)
spin_button.setValue(5)

# Подключаем обработчики сигналов
spin_button.valueChanged.connect(lambda val: print(f"Value selected: {val}"))
spin_button.clicked.connect(lambda: print("AnimatedButton clicked"))

main_widget.show()
app.exec_()
```

В этом примере виджет `AnimatedButton` добавляется в вертикальный layout обычного QWidget. При запуске будет видна только компактная кнопка с числом **5** (которое мы установили). При нажатии на неё развернётся числовой ввод (`QSpinBox`), где пользователь может изменить значение. Каждое изменение напечатает новое значение в консоль, а сам факт нажатия (раскрытия) также логируется.

**2. Кастомизация внешнего вида:**

```
spin_button = AnimatedButton()
spin_button.setValue(10)
# Настроим стиль: сделаем фон темным, текст светлым, ускорим анимацию
spin_button.apply_custom_style({
    'background_color': '#2b2b2b',  # тёмно-серый фон кнопки
    'text_color': '#f0f0f0',        # светло-серый текст
    'active_color': '#ff9900',      # оранжевый цвет акцента при фокусе/
нажатии
    'padding': 6,                   # чуть больше внутренний отступ
    'anim_speed': 200               # анимация немного быстрее (200 мс)
})
```

После такого вызова AnimatedButton будет стилизован под тёмную тему: в свёрнутом виде кнопка, вероятно, покажет число *10* светлым текстом на тёмном фоне. При наведении или фокусе может появиться оранжевая обводка/подсветка (active_color), анимация раскрытия/сворачивания займёт 0.2 секунды.

**3. Использование нескольких AnimatedButton:** AnimatedButton может использоваться в нескольких экземплярах одновременно, как и любой виджет. Например, можно создать массив таких кнопок для разных настроек:

```
options = ["Количество попыток", "Число элементов", "Макс. скорость"]
layout = QFormLayout()
for label in options:
    btn = AnimatedButton()
    btn.set_scale(0.9)
    layout.addRow(label + ":", btn)
```

В этом гипотетическом примере мы создаём три AnimatedButton, уменьшаем их масштаб немного (0.9) для компактности и располагаем с текстовыми метками в форме. Каждая кнопка может иметь своё значение. Они функционируют независимо, и все сигналы `valueChanged` можно привязать к разным обработчикам (или к одному универсальному с проверкой отправителя). Таким образом, AnimatedButton хорошо интегрируется в формы и панели настройки.

**Важно:** AnimatedButton – самостоятельный компонент. Он не требует специального контейнера или особого менеджера компоновки, достаточно добавить его в любой доступный `QLayout` вашего окна или диалога. Он будет занимать **минимальный размер** в свёрнутом виде и автоматически резервировать больше места при раскрытии. Однако, чтобы анимация выглядела корректно, убедитесь, что родительский layout может расширяться (например, `QHBoxLayout` или `QVBoxLayout` обычно справляются с этим). Если поместить AnimatedButton в контейнер фиксированного размера, при раскрытии он может не показать весь `QSpinBox`. Обычно же менеджеры компоновки сами подстроят окно под увеличившийся размер, либо, если в макете предусмотрено свободное пространство, AnimatedButton займет его.

## Логирование

В класс AnimatedButton встроено логирование действий для отладки. Компонент использует модуль стандартной библиотеки Python `logging` для выдачи сообщений о своём состоянии и событиях. Это **особенно полезно**, если что-то работает не так, как ожидалось – можно включить отладочный вывод и посмотреть, в какой последовательности происходят переходы состояний и вызовы методов.

Что логируется (на уровне DEBUG): - **Переходы состояний:** при вызове методов `expand_button()` и `collapse_button()` в лог пишется, что начато раскрытие или сворачивание. По завершении анимации может логироваться достижение состояния Expanded/Collapsed. Например: *"AnimatedButton: expanding (state=Expanding)"*, *"AnimatedButton: expanded (state=Expanded)"* и аналогичные сообщения для сворачивания. - **Нажатия и взаимодействия:** факт нажатия (сигнал `clicked`) может сопровождаться записью, например: *"AnimatedButton: clicked (initiating expand)"*. Если используются события потери фокуса для автозакрытия, это также может быть залогировано: *"AnimatedButton: focus out - collapsing"* (если реализовано автосворачивание при покидании фокуса). - **Изменения значения:** при эмиссии сигнала `valueChanged` компонент может выводить сообщение вида *"AnimatedButton: value changed to X"*, показывающее новое установленное значение. - **Применение стиля:** вызов `apply_custom_style` выводит в лог переданные пользователем параметры и итоговые значения после слияния с темой. Например, *"AnimatedButton: applying custom style {'padding': 8, 'text_color': '#ffffff'}"* и *"AnimatedButton: style applied (padding=8, text_color=#ffffff, ... остальные параметры)"*. Это помогает убедиться, что нужные настройки действительно установились. - **Масштабирование:** аналогично, при вызове `set_scale(…)` можно ожидать сообщение

*"AnimatedButton: scale set to 1.2"* и, возможно, дополнительные детали о том, какие размеры изменены.

По умолчанию уровень логирования может быть установлен на WARNING или INFO, поэтому debug-сообщения не выводятся. Чтобы их включить, необходимо настроить логгер. Сделать это можно в вашем приложении следующим образом:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Этот вызов сконфигурирует базовый логгер и выведет все сообщения уровня DEBUG и выше в stdout (консоль). Если вы хотите ограничить вывод логов только сообщениями AnimatedButton (не засоряя вывод сообщениями других библиотек), вы можете настроить логгер по имени модуля/класса. Предположим, AnimatedButton находится в модуле `animated_button.py` и внутри класса используется `logging.getLogger(__name__)`. Тогда можно сделать:

```
logger = logging.getLogger('animated_button')
logger.setLevel(logging.DEBUG)
```

Теперь отладочные сообщения, относящиеся только к AnimatedButton, будут видны.

**Пример логов при работе компонента:**

```
DEBUG:animated_button: AnimatedButton initialized (state=Collapsed, value=0)
DEBUG:animated_button: clicked -> start expanding
DEBUG:animated_button: expanding animation start (duration=300ms)
DEBUG:animated_button: value changed to 5
DEBUG:animated_button: expanding animation end (state=Expanded)
DEBUG:animated_button: collapse_button() called
DEBUG:animated_button: collapsing animation start (duration=300ms)
DEBUG:animated_button: collapsing animation end (state=Collapsed)
```

Как видно, по этим записям можно проследить весь цикл работы: инициализация, клик (начало раскрытия), изменение значения, завершение раскрытия, вызов сворачивания и его завершение. В реальности формулировки могут немного отличаться, но суть остаётся: логирование в AnimatedButton призвано помочь разработчику понять динамику поведения виджета и быстрее найти возможные проблемы.

Если требуется ещё более подробная информация, можно модифицировать исходный код класса, добавив дополнительные `logging.debug()` вызовы в интересующих местах, однако обычно встроенного логирования достаточно для анализа стандартных ситуаций. Не забудьте выключить или повысить уровень логирования на production-сборке приложения, чтобы лишние сообщения не захламляли логи пользователя.

---