**Асинхронное приложение ч.1 / tkinter 15**

14.11.2020

**Планирование действий**

Базовый метод предотвращения блокировки основного потока в Tkinter — это планирование действий, которые будут выполнены после истечения заданного времени.

В этом материале разберемся с тем, как реализовать этот подход в Tkinter с помощью метода after(), который может быть вызван во всех классах виджетов.

Следующий код показывает пример того, как функция обратного вызова может блокировать основной цикл.

Это приложение состоит из одной кнопки, которая становится неактивной после нажатия. Через 5 секунд ее снова можно нажать. Простейшая реализация будет выглядеть следующим образом:

КОПИРОВАТЬ

import time

import tkinter as tk

class App(tk.Tk):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.button = tk.Button(self, command=self.start\_action,

text="Ждать 5 секунд")

self.button.pack(padx=50, pady=20)

def start\_action(self):

self.button.config(state=tk.DISABLED)

time.sleep(5)

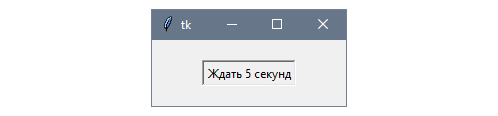
self.button.config(state=tk.NORMAL)

if \_\_name\_\_ == "\_\_main\_\_":

app = App()

app.mainloop()

Если запустить эту программу, то можно заметить, что не кнопка становится неактивной, а весь графический интерфейс зависает на 5 секунд. Это понятно по внешнему виду кнопки, которая в течение 5 секунд выглядит нажатой, а не выключенной. Более того, строка заголовка не будет реагировать на клики мыши все это время:



Если активировать дополнительные виджеты, например, Entry и Scroll, то это поведение задело бы и их.

А теперь посмотрим, как добиться нужного поведения вместо того, чтобы блокировать выполнение потока.

**Как работает планирование действий**

Метод after() позволяет регистрировать функцию обратного вызова, которая вызывается после задержки, заданной в миллисекундах в основном цикле Tkinter. По сути, они представляют собой зарегистрированные сигналы-события, которые обрабатываются в те моменты, когда система находится в состоянии ожидания.

Таким образом заменим вызов time.sleep(5) на self.after(5000,callback). Используем экземпляр self, потому что метод after() также доступен в корневом экземпляре Tk, и нет разницы в том, чтобы вызывать его из дочернего виджета:

КОПИРОВАТЬ

import tkinter as tk

class App(tk.Tk):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.button = tk.Button(self, command=self.start\_action,

text="Ждать 5 секунд")

self.button.pack(padx=50, pady=20)

def start\_action(self):

self.button.config(state=tk.DISABLED)

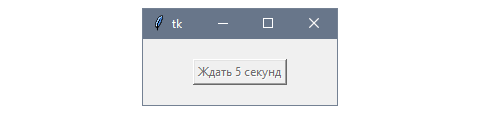
self.after(5000, lambda: self.button.config(state=tk.NORMAL))

if \_\_name\_\_ == "\_\_main\_\_":

app = App()

app.mainloop()

Благодаря этому приложение будет реагировать вплоть до запланированного действия. Кнопка станет неактивной, но со строкой заголовка можно продолжать взаимодействовать привычным образом:



По последнему примеру можно предположить, что метод after() исполняется после заданной в миллисекундах длительности.

Однако на самом деле метод просит Tkinter зарегистрировать событие, что гарантирует, что оно не выполнится ранее намеченного времени. И если основной поток занят, то верхнего предела того, когда оно все-таки выполнится, нет.

Нужно также помнить о том, что выполнение метода продолжается сразу же после планирования действия. Следующий пример иллюстрирует такое поведение:

КОПИРОВАТЬ

print("Первый")

self.after(1000, lambda: print("Третий"))

print("Второй")

Этот код выведет «Первый», «Второй» и, наконец, «Третий» спустя секунду. Все это время графический интерфейс будет оставаться доступным, а пользователи смогут продолжать взаимодействовать с ним.

Обычно нужно также не допустить, чтобы одно и то же фоновое задание выполнялось более одного раза, поэтому хорошей практикой считается отключение виджета, запустившего выполнение.

Не стоит забывать, что любая запланированная функция будет выполнена в основном потоке, поэтому одного только after() недостаточно, чтобы предотвратить зависание интерфейса. Нужно также не выполнять методы, выполнение которых занимает много времени в качестве обратного вызова.

В следующем примере рассмотрим, как можно сделать так, чтобы эти блокирующие действия выполнялись в отдельных потоках.

Метод after() возвращает идентификатор запланированного события, который можно передать в метод after\_cancel() для отмены выполнения функции обратного вызова.

Дальше рассмотрим, как реализовать остановку запланированного события с помощью этого метода.

**Работа в потоках**

Поскольку основной поток отвечает только за обновление графического интерфейса и обработку событий, оставшаяся часть фоновых событий должна выполняться на разных потоках.

[**Стандартная библиотека Python**](https://pythonru.com/biblioteki) включает модуль threading для создания и контроля несколько потоков с помощью высокоуровневого интерфейса, который позволяет работать с простыми классами и методами.

Стоит отметить, что CPython — «эталонная реализация» Python — ограничена GIL (Global Interpreter Lock), механизмом, который не дает нескольким потокам запускать байт-код Python одновременно. Из-за этого невозможно пользоваться преимуществами многопроцессорных систем. Об этом важно помнить при попытке улучшить производительность приложения.

В следующем примере объединены приостановка потока с помощью time.sleep(), а также действие, запланированное с помощью after():

КОПИРОВАТЬ

import time

import threading

import tkinter as tk

class App(tk.Tk):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.button = tk.Button(self, command=self.start\_action,

text="Ждать 5 секунд")

self.button.pack(padx=50, pady=20)

def start\_action(self):

self.button.config(state=tk.DISABLED)

thread = threading.Thread(target=self.run\_action)

print(threading.main\_thread().name)

print(thread.name)

thread.start()

self.check\_thread(thread)

def check\_thread(self, thread):

if thread.is\_alive():

self.after(100, lambda: self.check\_thread(thread))

else:

self.button.config(state=tk.NORMAL)

def run\_action(self):

print("Запуск длительного действия...")

time.sleep(5)

print("Длительное действие завершено!")

if \_\_name\_\_ == "\_\_main\_\_":

app = App()

app.mainloop()

**Как работают треды**

Для создания нового объекта Thread можно использовать конструктор и аргумент-ключевое слово target. Он будет вызван на отдельном потоке при использовании его же метода start().

В прошлом примере использовалась ссылка на метод run\_action, примененная экземпляру текущего приложения:

КОПИРОВАТЬ

thread = threading.Thread(target=self.run\_action)

thread.start()

После этого периодически опрашивается статус потока после after(), который планирует тот же метод до завершения потока:

КОПИРОВАТЬ

def check\_thread(self, thread):

if thread.is\_alive():

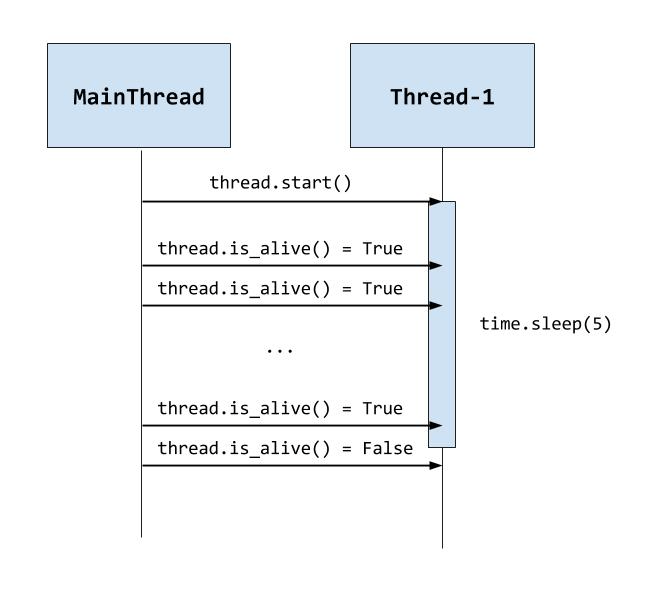
self.after(100, lambda: self.check\_thread(thread))

else:

self.button.config(state=tk.NORMAL)

В прошлом примере задержка была 100 миллисекунд, потому что чаще проверять статус нет необходимости. Хотя это всегда зависит от типа действия на потоке.

Это процесс может быть представлен в виде такой диаграммы:



Прямоугольник Thread-1 представляет время, во время которого поток занят выполнением time.sleep(5). В то же время MainThread только проверяет статус, и нет ни одной операции, которая приводила бы к зависанию всего интерфейса.

В этом материале мы познакомились с классом Thread, но важно остановиться на некоторых деталях создания их экземпляров и использования в программах на Python.

**Методы Thread — start, run и join**

start() в этом примере вызывался для выполнения метода в отдельном потоке, чтобы основной продолжал выполняться.

Если же вызвать join(), то основной был бы заблокирован до остановки нового. Это привело бы к тому же «зависанию», которого мы пытались избежать, даже при использовании нескольких потоков.

Наконец, метод run()— это то, где поток выполняет операцию. В будущем его нужно перезаписывать.

Важно запомнить, что нужно всегда вызывать start() из основного потока, чтобы не блокировать его.

**Параметры для метода**

При использовании конструктора класса Thread можно задать аргументы для передаваемого метода с помощью параметра args:

КОПИРОВАТЬ

def start\_action(self):

self.button.config(state=tk.DISABLED)

thread = threading.Thread(target=self.run\_action, args=(5,))

thread.start()

self.check\_thread(thread)

def run\_action(self, timeout):

# ...

Параметр selfпередается автоматически, поскольку используется текущий экземпляр для ссылки на переданный метод. Это удобно в тех ситуациях, когда новому потоку нужен доступ к информации из экземпляра вызвавшего его.