Передача сигналов и данных между объектами

Мод… …струю и простую передачу сигналов между объектами и широковещательную передачу сигналов для объ…

Ядр… …аленькое, но обладает мощными функциями:

- …нных сигналов;
- анонимные сигналы;
- пользовательские реестры имен;
- постоянно или временно подключенные приемники;
- автоматически отключенные приемники через слабую привязку;
- отправка произвольных полезных данных;
- сбор возвращаемых значений от приемников сигналов;
- потокобезопасность.

Содержание:

Установка модуля blinker в виртуальное окружение.

```
# создаем виртуальное окружение, если нет
$ python3 -m venv .venv --prompt VirtualEnv
# активируем виртуальное окружение
$ source .venv/bin/activate
# обновляем `pip`
(VirtualEnv):~$ python3 -m pip install -U pip
# ставим модуль `blinker`
(VirtualEnv):~$ python3 -m pip install blinker
```

Именованные сигналы модуля blinker.

Именованные сигналы создаются с помощью функции signal():

```
>>> from blinker import signal
>>> initialized = signal('initialized')
>> Вверх alized is signal('initialized')
# True
```

> *Каждый вызов signal('name') возвращает один и тот же объект сигнала, позволяя несвязанным частям кода (разным модулям, плагинам, чему угодно) использовать один и тот же сигнал, не требуя совместного использования кода или специального импорта.*

Метод Signal.connect(): подписка на сигналы blinker.

Метод Signal.connect(receiver, sender=ANY, weak=True) регистрирует функцию receiver, которая будет вызываться каждый раз при испускании сигнала. Связанным функциям всегда передается объект, вызвавший генерацию сигнала.

```
from blinker import signal

def subscriber(sender):
    print(f"Получил сигнал, посланный {sender!r}")

>>> ready = signal('ready')
>>> ready.connect(subscriber)
# <function subscriber at 0x7f03d0122550>
```

Метод Signal.connect() также принимает 2 дополнительных аргумента:

- sender=blinker.ANY - аргумент ограничивает уведомления, доставляемые получателю receiver, только теми сообщениями Signal.send(), которые отправлены отправителем. Если sender=blinker.ANY (по умолчанию), то получатель receiver всегда будет уведомлен. Получатель - sender=blinker.ANY receiver может быть подключен к нескольким значениям отправителя в одном и том же Signal через несколько вызовов Signal.connect().
- weak=True - если True, то сигнал будет содержать слабую ссылку на получателя и автоматически отключится, когда получатель выйдет за пределы области видимости или отработает сборщик мусора.

Метод Signal.send(): отправка сигналов blinker.

Код, создающий интересующие события, может отправлять уведомления Signal.send(*sender, **kwargs) всем подключенным получателям.

Принимаемые аргументы:

- *sender - любой объект или None. Если опущено, то синоним None. Принимает только один позиционный аргумент.
- **kwargs - данные для отправки получателям.

Ниже простой класс Processor() выдает сигнал готовности, когда он собирается что-то обработать, и завершается, когда это делается. Он передает self методу Processor.send(), показывая, что этот конкретный экземпляр был ответственен за отправку сигнала.

```
class Processor:
   def __init__(self, name):
       self.name = name

   def go(self):
       ready = signal('ready')
       ready.send(self)
       print("Обработка.")
       complete = signal('complete')
       complete.send(self)

   def __repr__(self):
       return f'<Processor {self.name}>'

>>> processor_a = Processor('a')
>>> processor_a.go()
# Получил сигнал, посланный <Processor a>
# Обработка.
```

Обратите внимание на сигнал 'complete' в методе Processor.go(). Пока ни один получатель не подключился к complete, и это нормально. Вызов метода Signal.send() для сигнала без получателей приведет к тому, что уведомления не будут отправлены, и эти неактивные отправки оптимизированы, чтобы быть как можно более дешевыми.

Подписка на определенных отправителей.

Подключение к сигналу по умолчанию вызывает функцию приемника, когда его посылает любой отправитель. Функция Signal.connect() принимает необязательный аргумент sender, для ограничения подписки одним конкретным отправляющим объектом:

```
de Вверх scriber(sender):
    print("Пойманный сигнал от processor_b.")
```

```
    assert sender.name == 'b'

>>> processor_b = Processor('b')
>>> ready.connect(b_subscriber, sender=processor_b)
# <function b_subscriber at 0x...>
```

Эта функция была подписана на готовность, но только при отправке processor_b:

```
>>> processor_a.go()
# Получил сигнал, посланный <Processor a>
# Обработка.
>>> processor_b.go()
# Получил сигнал, посланный <Processor b>
# Пойманный сигнал от processor_b.
# Обработка.
```

Отправка и получение данных через сигналы blinker.

В метод Signal.send() могут быть переданы дополнительные ключевые аргументы **kwargs. Они, в свою очередь, будут переданы подключенным функциям:

```
send_data = signal('send-data')

@send_data.connect
def receive_data(sender, **kw):
    print(f'Пойманный сигнал от {sender!r}, data {kw!r}")
    return 'received!'

>>> result = send_data.send('anonymous', abc=123)
# Пойманный сигнал от 'anonymous', data {'abc': 123}
>>> result
# [(<function receive_data at 0x7f03d0240280>, 'received!')]
```

Возвращаемое значение метода Signal.send собирает возвращаемые значения каждой подключенной функции в виде списка пар (receiver_function, return_value):

Анонимные сигналы модуля blinker.

Сигналам не нужно давать названия. Конструктор объекта Signal создает уникальный сигнал при каждом вызове. Например, альтернативная реализация вышеописанного класса Processor может предоставлять сигналы обработки в виде атрибутов класса:

```
from blinker import Signal

class AltProcessor:
   on_ready = Signal()
   on_complete = Signal()

   def __init__(self, name):
       self.name = name

   def go(self):
       self.on_ready.send(self)
       print("Альтернативная обработка.")
       self.on_complete.send(self)

   def __repr__(self):
       return f'<AltProcessor {self.name}>'
```

Использование метода Signal.connect в качестве декоратора.

В разделах выше можно было видеть возвращаемое значение метода Signal.connect(). Это позволяет использовать Signal.connect() в качестве декоратора для функций:

```
apc = AltProcessor('C')

@apc.on_complete.connect
def completed(sender):
    print f"AltProcessor {sender.name} завершен!"

>>> apc.go()
# Альтернативная обработка.
# AltProcessor C завершен!
```

Вверх

Несмотря на удобство, эта форма, к сожалению, не позволяет настроить аргументы sender или weak которые необходимы для подключенной функции. Для этого можно использовать метод Signal.connect_via():

```
dice_roll = signal('dice_roll')
                          ⋮
@dice_roll.connect_via(1)
@dice_roll.connect_via(3)
@dice_roll.connect_via(5)
def odd_subscriber(sender):
    print(f"Наблюдаемый бросок костей {sender!r}.")

>>> result = dice_roll.send(3)
# Наблюдаемый бросок костей 3.
```

Оптимизация отправки сигналов blinker.

Сигналы оптимизированы для очень быстрой отправки независимо от того, подключены ли приемники или нет. Если ключевые аргументы (данные), которые должны быть отправлены с сигналом, требуют больших затрат для вычисления, то может быть более эффективным сначала проверить, подключены ли какие-либо получатели, проверив свойство Signal.receivers:

```
>>> bool(signal('ready').receivers)
# True
>>> bool(signal('complete').receivers)
# False
>>> bool(AltProcessor.on_complete.receivers)
# True
```

Также возможна проверка получателя, прослушивающего конкретного отправителя:

```
>>> signal('ready').has_receivers_for(processor_a)
# True
```


Вверх