Сообщить об ошибке.

Модуль fire в Python, автоматическое создание CLI

Создания CLI с помощью одной строки кода

Модуль fire представляет собой инструмент для автоматического создания интерфейсов командной строки (CLI) с помощью одной строки кода. Она превратит любой модуль Python, класс, объект, функцию и т.д. в CLI (любой компонент Python будет работать!).

Установка модуля fire в виртуальное окружение.

Модуль fire размещен на PyPI, поэтому установка относительно проста.

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

Содержание:

Основы работы с модулем fire.

Самый простой способ использовать Fire - взять любую программу на Python, а затем просто вызвать fire.Fire() в конце программы. Это откроет полное содержимое программы в командной строке.

```python
# test.py
import fire

def hello(name):
  return f'Hello {name}!'

if __name__ == '__main__':
  fire.Fire()
```

Вывод справки при запуске из командной строки:

```
$ python3 test.py --help
NAME
    test.py
```

```
SYNOPSIS
    test.py GROUP | COMMAND
GROUPS
    GROUP is one of the following:
     fire
       The Python Fire module.
COMMANDS
    COMMAND is one of the following:
     hello
```

Командой COMMAND будет служить имя функции hello(). Вот как можно запустить саму программу из командной строки:

```
$ python3 test.py hello World
Hello World!
```

Немного изменим программу, чтобы функция hello отображалась только в командной строке.

```python
# test.py
import fire

def hello(name):
  return f'Hello {name}!'

if __name__ == '__main__':
    # передаем в качестве аргумента
    # имя функции hello()
    fire.Fire(hello)
```

Пример справки:

```
$ python3 test.py --help
NAME
    test.py
SYNOPSIS
    test.py NAME
POSITIONAL ARGUMENTS
    NAME
NOTES
    You can also use flags syntax for POSITIONAL ARGUMENTS
```

Запускаем программу из командной строки:

```
$ python3 test.py World
Hello World!
```

Обратите внимание, что при запуске теперь больше не надо указывать команду hello, т. к. ее имя передается в fire.Fire(hello).

В качестве альтернативы, можно написать эту программу следующим образом:

```python
# test.py
import fire

def hello(name):
  return f'Hello {name}!'

def main():
  fire.Fire(hello)

if __name__ == '__main__':
  main()
```

Также, если есть файл test.py, и при этом не хочется импортировать модуль fire:

```python
# test.py
def hello(name):
  return f'Hello {name}!'
```

То из командной строки этот файл можно использовать следующим образом:

```
# запуск скрипта `test.py`
$ python3 -m fire test.py hello --name=World
Hello World!
# запуск скрипта как модуль `test`
$ python3 -m fire test hello --name=World
Hello World!
```

Очень здорово помогает, если нужно *по быстрому* протестировать функцию.

Строка `-m fire test hello --name=World` означает:

- `-m`: команда Python - подключить модуль;
- `fire`: имя подключаемого модуля;
- `test`: путь к скрипту Python или имя запускаемого модуля (указывается без расширения .py) ;
- `hello`: имя функции в скрипте Python;
- `--name=World`: аргументы функции.

Предоставление нескольких команд в CLI.

В предыдущем примере, командной строке была представлена одна функция. Теперь рассмотрим способы представления нескольких функций в командной строке.

Самый простой способ представить несколько команд - написать несколько функций, а затем вызвать `fire.Fire()`.

```
# test.py
import fire

def add(x, y):
  return x + y

def multiply(x, y):
  return x * y

if __name__ == '__main__':
  fire.Fire()
```

Можно использовать это так:

```
$ python3 test.py add 10 20
30
$ python3 test.py multiply 10 20
200
```

Заметили, что модуль `fire` правильно разобрал 10 и 20 как числа, а не как строки. Подробнее о разборе параметров CLI читайте ниже.

Предыдущий код представляет всю функциональность программы в командной строке. Используя словарь, можно выборочно отображать функции в командной строке.

```
# test.py
import fire

def add(x, y):
  return x + y

def multiply(x, y):
  return x * y

if __name__ == '__main__':
  fire.Fire({
      'add': add,
  })
```

Вывод справки и пример выполнения:

```
$ python3 test.py --help
NAME
    test.py
SYNOPSIS
    test.py COMMAND
COMMANDS
    COMMAND is one of the following:
     add

$ python3 test.py add 10 20
30
```

Модуль fire также действует на объекты. Это хороший способ предоставить CLI несколько команд.

```
# example.py
import fire

class Calculator:

  def add(self, x, y):
    return x + y

  def multiply(self, x, y):
    return x * y

if __name__ == '__main__':
  calculator = Calculator()
  fire.Fire(calculator)
```

Можно использовать так же, как и раньше:

```
$ python3 example.py add 10 20
30
$ python3 example.py multiply 10 20
200
```

Модуль fire также работает с классами. Это еще один хороший способ предоставить CLI несколько команд.

```
import fire

class Calculator:

  def add(self, x, y):
    return x + y

  def multiply(self, x, y):
    return x * y

if __name__ == '__main__':
  fire.Fire(Calculator)
```

Почему можно предпочесть класс объекту? Одна из причин заключается в том, что также можно передавать аргументы для построения класса, как в следующем примере с BrokenCalculator.

```
import fire

class BrokenCalculator:

  def __init__(self, offset=1):
      self._offset = offset

  def add(self, x, y):
    return x + y + self._offset

  def multiply(self, x, y):
    return x * y + self._offset
```

```
if __name__ == '__main__':
  fire.Fire(BrokenCalculator)
```

Если использовать "*сломанный калькулятор*", то получим неправильные ответы:

```
$ python3 example.py add 10 20
31
$ python3 example.py multiply 10 20
201
```

Но это всегда можно исправить:

```
$ python3 example.py add 10 20 --offset=0
30
$ python3 example.py multiply 10 20 --offset=0
200
```

В отличие от вызова обычных функций, которые могут быть выполнены как с позиционными аргументами, так и с именованными аргументами (синтаксис --flag), следовательно аргументы в методе __init__ то же должны передаваться с синтаксисом --flag. Дополнительную информацию смотрите в разделе о вызове функций.

Группировка команд для предоставления в CLI.

Пример того, как можно создать интерфейс командной строки CLI со сгруппированными командами.

```
class IngestionStage:

  def run(self):
    return 'Ingesting! Nom nom nom...'

class DigestionStage:

  def run(self, volume=1):
    return ' '.join(['Burp!'] * volume)

  def status(self):
    return 'Satiated.'

class Pipeline:

  def __init__(self):
    self.ingestion = IngestionStage()
    self.digestion = DigestionStage()

  def run(self):
    self.ingestion.run()
    self.digestion.run()
    return 'Pipeline complete'

if __name__ == '__main__':
  fire.Fire(Pipeline)
```

Можно вкладывать свои команды произвольно сложными способами.

Вот как это выглядит в командной строке:

```
$ python3 example.py run
Ingesting! Nom nom nom...
Burp!
$ python3 example.py ingestion run
Ingesting! Nom nom nom...
$ python3 example.py digestion run
Burp!
$ python3 example.py digestion status
Satiated.
```

Доступ к свойствам объектов с модулем fire.

В примерах, которые рассмотрели выше, все вызовы python3 example.py запускали некоторую функцию/метод из примера программы. В следующем примере попробуем обратиться к свойству.

Примечание: Сторонний модуль airports позволяет искать аэропорт по 3-буквенному коду IATA. Чтобы запустить следующий пример, необходимо сначала установить этот модуль в виртуальное окружение командой python3 -m pip install airports.

```
from airports import airports
import fire

class Airport:

  def __init__(self, code):
    self.code = code
    self.name = dict(airports).get(self.code)
    self.city = self.name.split(',')[0] if self.name else None

if __name__ == '__main__':
  fire.Fire(Airport)
```

Теперь можно использовать эту программу, чтобы узнать коды аэропортов!

```
$ python3 example.py --code=JFK code
JFK
$ python3 example.py --code=SJC name
San Jose-Sunnyvale-Santa Clara, CA - Norman Y. Mineta San Jose International (SJC)
$ python3 example.py --code=ALB city
Albany-Schenectady-Troy
```

Цепочки вызова параметров в CLI.

Когда запускается fire.Fire() CLI, то можно выполнять все те же действия с результатом вызова Fire, которые можно выполнять с переданным исходным объектом. Проще говоря, можно использовать все методы объекта, в качестве дополнительных параметров CLI, который возвращает работающая функция/метод.

Например, можно использовать интерфейс командной строки класса Airport из предыдущего примера следующим образом:

```
$ python3 example.py --code=ALB city upper
ALBANY-SCHENECTADY-TROY
```

Здесь используется метод str.upper(), в качестве дополнительного параметра, для вывода результата в верхнем регистре, так как этот метод является методом всех строк.

Так что, если необходимо, чтобы параметры CLI красиво выстраивались в цепочку, то все, что нужно сделать, это создать класс, методы которого возвращают self.

Вот пример:

```
import fire

class BinaryCanvas:
  """A canvas with which to make binary art, one bit at a time."""

  def __init__(self, size=10):
    self.pixels = [[0] * size for _ in range(size)]
    self._size = size
    self._row = 0  # The row of the cursor.
    self._col = 0  # The column of the cursor.

  def __str__(self):
    return '\n'.join(' '.join(str(pixel) for pixel in row) for row in self.pixels)

  def show(self):
    print(self)
```

```
    return self

  def move(self, row, col):
    self._row = row % self._size
    self._col = col % self._size
    return self

  def on(self):
    return self.set(1)

  def off(self):
    return self.set(0)

  def set(self, value):
    self.pixels[self._row][self._col] = value
    return self

if __name__ == '__main__':
  fire.Fire(BinaryCanvas)
```

Теперь можно нарисовать смайлик.

```
$ python3 example.py move 3 3 on move 3 6 on move 6 3 on move 6 6 on move 7 4 on move 7 5 on
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 1 0 0 0
0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Еще более простой пример CLI.

```
import fire
en = 'Hello World'
ru = 'Привет Мир'
fire.Fire()
```

Можно использовать его следующим образом:

```
$ python3 example.py en
Hello World
$ python3 example.py ru
Привет Мир
```

Правила вызова методов класса/функций.

Аргументы конструктору передаются по имени с использованием синтаксиса флага --name=value.

Например, рассмотрим этот простой класс:

```
import fire

class Building:

  def __init__(self, name, stories=1):
    self.name = name
    self.stories = stories

  def climb_stairs(self, stairs_per_story=10):
    for story in range(self.stories):
      for stair in range(1, stairs_per_story):
        yield stair
      yield 'Phew!'
```

```
    yield 'Done!'

if __name__ == '__main__':
  fire.Fire(Building)
```

Можно вызывать его следующим образом: python3 example.py --name='Sherrerd Hall'. Аргументы другим функциям могут передаваться позиционно или по имени с использованием синтаксиса флагов.

Для создания экземпляра Building и последующего запуска метода climb_stairs допустимы все следующие команды:

```
$ python3 example.py --name="Sherrerd Hall" --stories=3 climb_stairs 10
$ python3 example.py --name="Sherrerd Hall" climb_stairs --stairs_per_story=10
$ python3 example.py --name="Sherrerd Hall" climb_stairs --stairs-per-story 10
$ python3 example.py climb-stairs --stairs-per-story 10 --name="Sherrerd Hall"
```

Можно заметить, что:

- Дефисы и символы подчеркивания (- и _) взаимозаменяемы в именах параметров CLI и именах флагов.
- Аргументы конструктора могут идти после аргументов функции или перед ней.
- Знак равенства между именем флага и его значением не является обязательным.

Использование в функциях *args и **kwargs.

Модуль fire поддерживает функции, которые принимают *args или **kwargs.

Вот пример:

```
import fire

def order_by_length(*items):
  """Упорядочивает по длине, в алфавитном порядке."""
  sorted_items = sorted(items, key=lambda item: (len(str(item)), str(item)))
  return ' '.join(sorted_items)

if __name__ == '__main__':
  fire.Fire(order_by_length)
```

Запускаем:

```
$ python3 example.py dog cat elephant
cat dog elephant
```

Для указания конца аргументов вызываемой функции в CLI, необходимо использовать разделитель. Все аргументы после разделителя будут использованы для обработки результата функции, а не переданы самой функции. Разделителем по умолчанию является дефис -.

Вот пример, где используется разделитель.

```
$ python3 example.py dog cat elephant - upper
CAT DOG ELEPHANT
```

Без разделителя - *upper* считался бы еще одним аргументом.

```
$ python3 example.py dog cat elephant upper
cat dog upper elephant
```

Можно изменить разделитель с помощью флага --separator. Флаги всегда отделяются от команды изолированным --. Вот пример, где меняется разделитель.

```
$ python3 example.py dog cat elephant X upper -- --separator=X
CAT DOG ELEPHANT
```

Синтаксический анализ параметров CLI.

Типы аргументов определяются их значениями, а не сигнатурой функции, в которой они используются. Вы можете передать любой литерал Python из командной строки: числа, строки, кортежи, списки, словари (множества поддерживаются только в некоторых версиях Python). Также можно произвольно вкладывать коллекции, если они содержат только литералы.

Чтобы продемонстрировать это, создадим небольшой пример программы, которая сообщает тип любого аргумента, который ей передается:

```
import fire
fire.Fire(lambda obj: type(obj).__name__)
```

Смотрим результаты распознавания типа переданного параметра через CLI:

```
$ python3 example.py 10
int
$ python3 example.py 10.0
float
$ python3 example.py hello
str
$ python3 example.py '(1,2)'
tuple
$ python3 example.py [1,2]
list
$ python3 example.py True
bool
$ python3 example.py {name:David}
dict
```

В последнем примере простые слова автоматически заменяются строками.

Будьте внимательны с кавычками! Если нужно передать строку "10", а не целое число 10, то необходимо либо экранировать, либо заключать в число в кавычки. В противном случае командная строка bash съест кавычки и передаст программе Python 10 без кавычек.

```
$ python3 example.py 10
int
$ python3 example.py "10"
int
$ python3 example.py '"10"'
str
$ python3 example.py "'10'"
str
$ python3 example.py \"10\"
str
```

Необходимо всегда помнить, что сначала bash обрабатывает параметры командной строки, а затем модуль fire анализирует результат. Если необходимо передать в программу dict{'name': 'David Bieber'}, то можно попробовать следующее:

```
$ python3 example.py '{"name": "David Bieber"}'
dict
$ python3 example.py {"name":'"David Bieber"'}
dict
# Неправильно, анализируется как строка.
$ python3 example.py {"name":"David Bieber"}
str
# Неправильно, это даже не рассматривается как отдельный аргумент.
$ python3 example.py {"name": "David Bieber"}
<error>
```

Маркеры True и False анализируются как логические значения.

Можно указать логические значения с помощью синтаксиса флагов --name и --noname, которые устанавливают для имени значения True и False соответственно.

Продолжая предыдущий пример:

```
$ python3 example.py --obj=True
bool
$ python3 example.py --obj=False
bool
$ python3 example.py --obj
bool
$ python3 example.py --noobj
bool
```

Будьте осторожны с bool флагами! Если за флагом, который должен быть логическим, сразу следует токен, отличный от другого флага, то флаг примет значение токена, а не логическое значение. Можно решить эту проблему: поставив разделитель после последнего флага, явно указав значение логического флага (например, --obj=True) или убедившись, что после любого аргумента логического флага есть еще один флаг.

Использование флагов модуля fire.

Все интерфейсы командной строки CLI модуля fire поставляются с рядом флагов. Эти флаги должны быть отделены от команды изолированным символом --. Если в CLI имеется хотя бы один изолированный --, то параметры после последнего изолированного -- рассматриваются как флаги, тогда как все аргументы *до последнего* -- считаются частью команды CLI.

Одним из полезных флагов является флаг --interactive (запуск в интерактивном режиме). Используйте флаг --interactive в любом интерфейсе командной строки, чтобы войти в Python REPL со всеми модулями и переменными, используемыми в контексте, где был вызван fire.Fire(). Также будут доступны другие полезные переменные, такие как результат команды Fire. Используйте этот флаг следующим образом: python3 example.py --interactive.

Можно добавить флаг --help к любой команде, чтобы просмотреть справку и информацию об использовании. Модуль fire включает информацию об использовании в строки документации, которые он генерирует. Модуль fire попытается вывести справку, даже если -- опущены, но не всегда может это сделать, так как строка help может являться допустимым именем аргумента. Используйте эту функцию следующим образом: python example.py -- --help или python example.py --help или даже python example.py -h.

Полный набор доступных флагов:

- command -- --help: справка и информация об использовании для команды.
- command -- --interactive: входит в интерактивный режим.
- command -- --separator=X: устанавливает разделитель на X. Разделителем по умолчанию является -.
- command -- --completion [shell]: создает скрипт завершения для CLI.
- command -- --trace: получает трассировку fire для команды.
- command -- --verbose: включает закрытые элементы в вывод.