Отчёт по летней практике в команде PyCharm

Выполнил: Александр Верещагин
Руководитель: Андрей Власовских

Введение

Язык программирования Python является динамически-типизированным скриптовым языком программирования, сочетающим в себе парадигмы объектно-ориентированного и функционального программирования. Он используется главным образом для разработки веб-приложений (например, с помощью Django) и решения задач системного администрирования (например, менеджер пакетов в Gentoo). В последнее время Python всё чаще используется как средство для научных вычислений в качестве альтернативы среде Matlab и подобным продуктам. К основными библиотеками, используемыми для научных вычислений, относятся Numpy, Matplotlib и Scipy.

Одним из продуктов компании JetBrains является интегрированная среда разработки на Python — PyCharm. На момент прохождения практики (версия 2.5) в PyCharm отсутствует специализированные средства поддержки библиотек для научных вычислений. Цель летней практики заключалась в реализации поддержки научных вычислений на языке Python в PyCharm. Результатом практики стал плагин, добавляющий некоторую поддержку для научных вычислений в PyCharm.

Поддержка типизации функций

Поскольку Python — динамически-типизированный язык программирования, функция может принимать и возвращать любой тип. Однако принимаемые и возвращаемый типы библиотечных функций известны заранее. Это позволяет выполнять проверку типов для таких функций средствами среды разработки. В PyCharm информация о типах библиотечных функций выделяется из строк документации (форматы Epydoc и reST), а для стандартной библиотеки — из заранее подготовленной базы данных. К сожалению, поддержка типов не работает для функций из рассматриваемых библиотек, поскольку библиотека Numpy использует собственный формат документации, в котором информация о типах содержится в разметке reST, а в других библиотеках, таких как Matplotlib, документация представляет собой словесное описание без указания информации о типах.

Поскольку большая часть документации библиотеки Numpy написана в единообразном стиле, имело смысл реализовать поддержку формата документации для библиотеки Numpy. Также, учитывая то, что для других библиотек невозможно выделить информацию о типах автоматически, было решено предоставить возможность добавлять информацию о типах вручную.

Описание формата документации Numpy

Документация Numpy представляет собой текст, размеченный в соответствии со стандартом reST. Информация о параметрах и типах содержится в разметке.

В составе библиотеки Numpy имеется встроенный парсер для строк документации. Парсер представляет собой независимый скрипт, написанный на языке Python, использующий регулярные выражения. Однако использовать этот парсер в плагине не представляется возможным, поскольку критически важно время работы парсера, а запуск интерпретатора Python достаточно длительный процесс.

Как составная часть плагина, был реализован подобный парсер, распознающий описание функции, типы и описание параметров и возвращаемого значения.

Строка документации состоит из следующих частей:

- Сигнатура функции. Для большинства функций сигнатура сдвинута на отступ влево относительно основной документации. По этой причине документация этих функций не распознаётся даже стандартным парсером. Данная особенность была учтена при разработке собственного парсера документации. После сигнатуры следует пустая строка.

- Описание функции. Может состоять из нескольких строк. После описания следует пустая строка. Далее может находиться несколько необязательных разделов документации. Раздел документации представляет собой заголовок раздела, после которого следует строка такой же длины, содержащая символы - или =.

- Раздел Parameters. В этом разделе содержится информация о типах параметров и их описание. Для каждого параметра первая строчка содержит название параметра и его тип, разделённые двоеточием. В качестве типа может использоваться название встроенного типа или типа библиотеки Numpy, кроме того множество допустимых значений, которые может принимать параметр, перечисленное в фигурных скобках, либо словесное описание типа. Если параметр необязателен, через запятую после типа следует слово optional. На следующих нескольких строчках с отступом вправо может находиться описание параметра.

- Раздел Returns. В этом разделе содержится информация о возвращаемом значении. Формат аналогичен формату раздела Parameters. Если раздел содержит больше одной записи, это означает, что функция возвращает кортеж из этих элементов.

Особенности документации Numpy

Документация для некоторых методов класса ndarray представляет собой ссылку на документацию соответствующей функции NumPy верхнего уровня.

В качестве примера ниже представлена строка документации метода *numpy.ndarray.sum*:

```
a.sum(axis=None, dtype=None, out=None)

Return the sum of the array elements over the given axis.

Refer to ‘numpy.sum‘ for full documentation.

See Also
--------
numpy.sum : equivalent function
```

То есть информацию о типах параметров этого метода нужно искать в документации к функции *numpy.sum*. Эта функция полностью аналогична приведённому методу с тем лишь исключением, что первым параметром она явно принимает экземпляр класса ndarray вместо неявного self.

Данная проблема была решена следующим образом: если в описании функции есть строка, удовлетворяющая регулярному выражению

```
^Refer to ‘(.*)‘ for full documentation.$
```

то для выделения информации о типах используется строка документации функции, на которую указывает ссылка.

Кроме того, документация для конструктора содержится в строке документации класса.

Формат хранения информации о типах

Информация о типах, выделяемая из строк документации, может быть получена динамически, в то время как заранее заданную информацию о типах нужно каким-то образом хранить. В ходе работы над плагином были рассмотрены следующие варианты форматов хранения:

- Java properties. Этот формат используется в PyCharm для хранения информации о типах для стандартных библиотек. Файл такого формата легко загрузить, он имеет текстовое представление, поэтому есть возможность редактирования вручную. Недостаток — отсутствие стандартных механизмов сериализации.
- XML-сериализация с помощью точки расширения applicationService. Эта точка расширения позволяет создавать контейнеры для автоматической сериализации и загрузки данных. Она используется платформой IDEA для хранения настроек среды. Однако загрузка и сохранение этого формата полностью автоматизированы и скрыты от разработчика. По этой причине довольно проблематично поддерживать такой формат.
- JDBM2, свободный аналог Berkley-DB для Java. Простое, быстрое и эффективное key-value хранилище. Данные хранятся в бинарном формате, поэтому редактирование базы данных вручную невозможно.
- JSON, библиотека Google-JSON включена в поставку платформы IDEA. Библиотека предоставляет возможность простой сериализации и десериализации. Данные хранятся в текстовом представлении.

Было решено остановиться на использовании формата JSON, поскольку он хорошо стандартизирован, позволяет редактировать информацию как вручную, так и автоматически при помощи библиотеки Google-JSON.

Ниже приведён пример файла с базой данных для типизированных функций в формате, который используется в плагине:

```
/* Список функций */
[
  {
    /* Имя функции */
    "name": "numpy.core.multiarray.arange",
    /* Возвращаемый тип */
    "returnType": "numpy.core.multiarray.ndarray",
    /* Список параметров */
    "parameters": [
      {
        /* Имя параметра */
        "name": "start",
        /* Тип параметра */
        "type": "int or long or float",
        /* Список допустимых значений или [] */
```

```
        "permissibleValues": [1, 2, 3]
      },
      /* Остальные параметры */
    ]
  },
  /* Остальные функции */
]
```

Динамические типы Numpy

Основным типом для хранения данных в библиотеке Numpy является тип ndarray, определяющий многомерный массив. Существует возможность явно задать тип элементов такого массива. Для этого при создании массива ndarray в качестве параметра для типа элементов используется экземпляр класса numpy.dtype, в конструктор которого передан один из следующих типов из пространства имён numpy:

```
int8, uint8, int16, uint16, int32, uint32, int64, uint64, int128, uint128,
float16, float32, float64, float80, float96, float128, float256,
complex32, complex64, complex128, complex160, complex192, complex256, complex512
```

Кроме того эти типы можно комбинировать, создавая записи с полями разных типов:

```
numpy.dtype([('f1', numpy.float80), ('f2', numpy.int32)])
```

Перечисленные выше типы создаются библиотекой Numpy динамически. По этой причине они не распознаются в PyCharm и подсвечиваются как предупреждение в инспекции PyUnresolvedReferencesInspection. Тем не менее точно известно, что эти типы существуют на момент выполнения скрипта. Чтобы избежать ложного срабатывания инспекции, было решено искусственно добавить данные типы в codeInsight с помощью точки расширения pyModuleMembersProvider.

Автодополнение для аргументов функций

Некоторые функции из библиотек Numpy и Matplotlib принимают аргументы из определённого множества значений. К примеру, функция sort из библиотеки Numpy определена следующим образом:

```
numpy.sort(a, axis=-1, kind='quicksort', order=None)
```

В качестве параметра kind функция может принимать одно из следующих значений:

- 'quicksort'
- 'mergesort'
- 'heapsort'

Несмотря на то, что TypeChecker не выдаст ошибку, если в качестве этого параметра будет передано какое-либо другое строковое значение, такой код будет являться некорректным. Поэтому было решено реализовать специализированный TypeChecker для подобных функций.

Кроме того пользователю было бы удобно, если бы при вводе аргументов для такого метода выдавались варианты автодополнения.

Для этого был реализован класс PermissibleArgumentCompletionContributor. Клас реализует точку расширения CompletionContributor и предлагает варианты дополнения для подобных функций в зависимости от того, в каком месте находится каретка.

Пользовательский интерфейс

Чтобы предоставить пользователю возможность добавлять информацию о типах для произвольной функции, был разработан соответствующий пользовательский интерфейс. Добавить типы для функции можно, вызвав контекстное меню на вызове функции и выбрав пункт "Edit type information...".

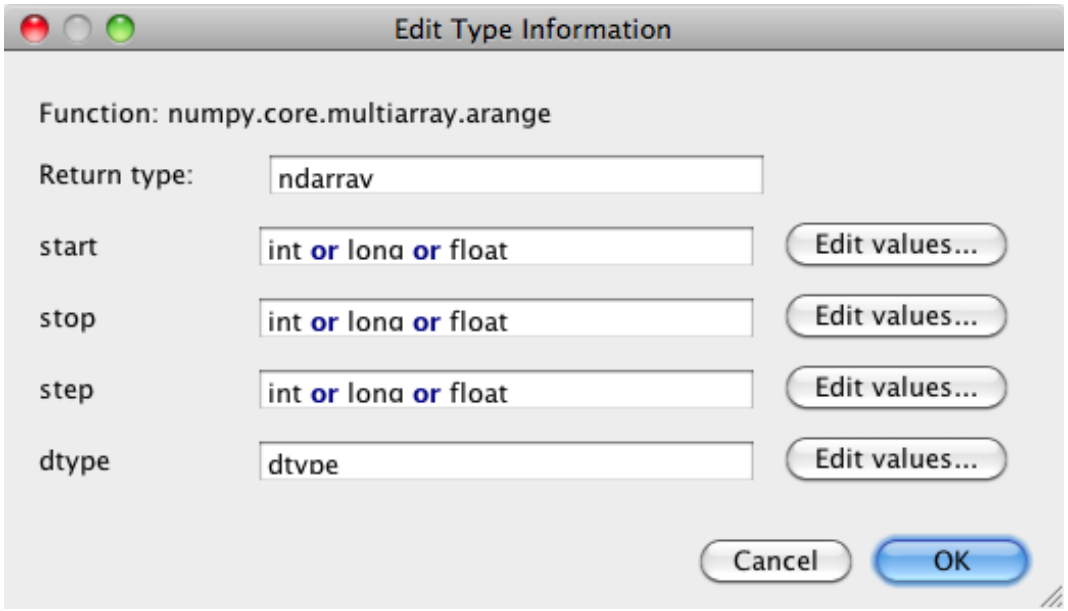

Рис. 1: Диалоговое окно Edit Type Information

В этом же окне можно перечислить допустимые значения аргументов, нажав на кнопку "Edit values...".

Кроме того пользовательскую базу данных типизированных функций можно редактировать через окно настроек PyCharm. Для этого была разработана панель настроек "User-Defined Type Information". Все функции, имеющиеся в базе данных отсортированы в алфавитном порядке, можно отфильтровать часть функций с помощью регулярного выражения. Функциональность панели позволяет удалять функции из базы данных и редактировать уже имеющиеся в ней функции.

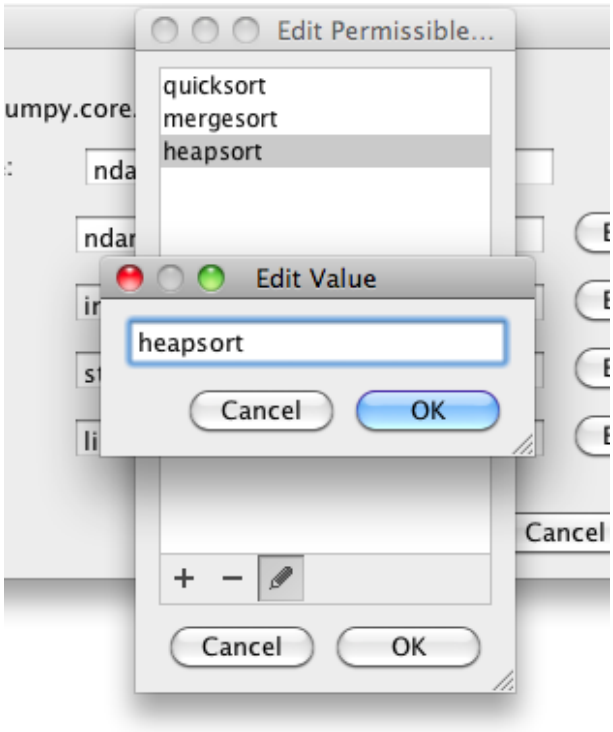

Рис. 2: Окно редактирования допустимых значений аргумента

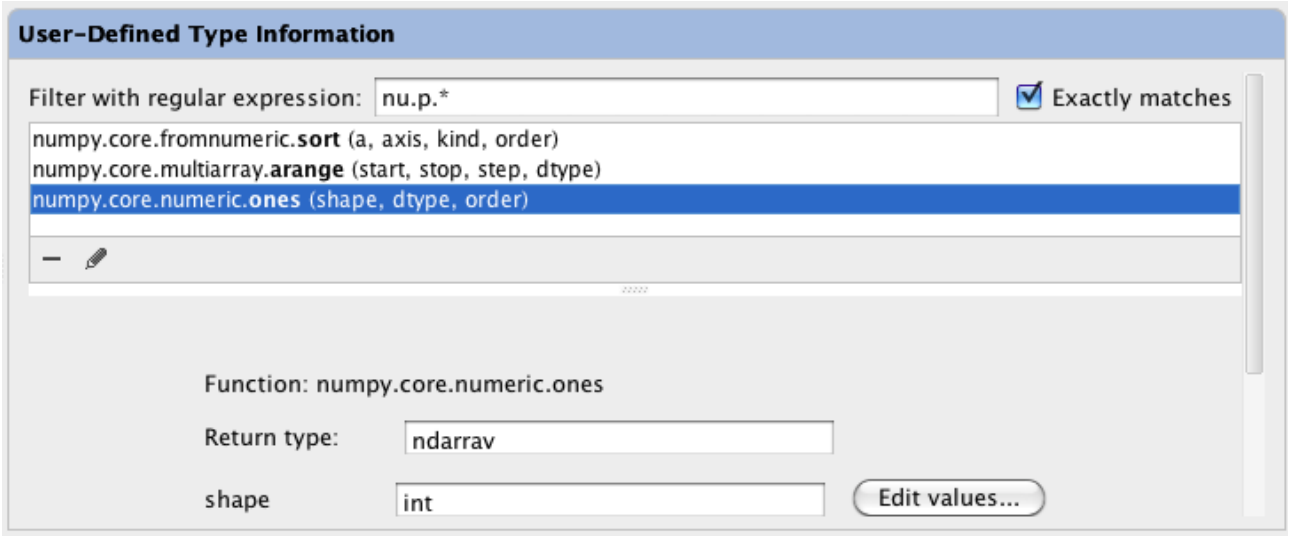

Рис. 3: Панель настроек User-Defined Type Information

Сборка плагина при отсутствии исходных кодов PyCharm

Репозиторий плагина доступен по адресу: `git://github.com/avereshchagin/pyscicomp.git`. Для корректной работы плагина требуется PyCharm с номером сборки не менее 121.160 (2.6 EAP).

1. Скопируйте репозиторий проекта на локальный диск:

   ```
   git clone git://github.com/avereshchagin/pyscicomp.git pyscicomp
   ```

2. В IntelliJ IDEA создайте новый проект без создания модуля.

3. В окне Project Structure выберите панель настроек Platform Settings/SDKs и добавьте новый IntelliJ IDEA Plugin SDK, указав путь к сборке PyCharm. Назовите его PyCharm SDK.

4. Перейдите на панель настроек Project Settings/Project и выберите в качестве Project SDK созданный в предыдущем пункте PyCharm SDK.

5. Перейдите на панель настроек Project Settings/Modules и импортируйте существующий модуль (Import existing module), выбрав файл pyscicomp.iml из репозитория плагина.

6. Нажмите OK.

7. В функциональном меню нажмите Edit Run Configurations и создайте новую конфигурацию запуска Plugin.

8. Плагин готов к сборке и запуску.

Возможности дальнейшего развития плагина

За время, отведённое на практику, удалось сделать не всё из того, что было запланировано. К функциональности плагина можно добавить следующие возможности:

- Выводить графики прямо в консоль Python. В силу особенностей реализации консоли для платформы IDEA, добавление такой функциональности на данный момент невозможно.
- Добавить поддержку нескольких консолей.
- Добавить окно для просмотра текущих переменных и истории команд.
- При запуске научной консоли можно автоматически импортировать необходимые библиотеки.
- На момент выполнения программы известны типы текущих переменных. Автодополнение можно выполнять в соответствии с действительным типом переменной на данный момент.
- Можно реализовать подсветку синтаксиса для кода в консоли Python.
- Можно добавить возможность сохранения истории консоли или её части в файл.
- Можно реализовать графический редактор переменных по аналогии с Matlab. Поскольку основной тип данных, как в Matlab, так и в Numpy — многомерный массив, чаще всего двумерный, элементы этого массива удобно редактировать в виде таблицы. Удобно также строить графики для заданных строк или столбцов, выделять подматрицы и фильтровать данные.

Заключение

В результате работы над плагином были использованы следующие точки расширения:

- LocalInspection: PermissibleArgumentCheckInspection — инспекция для проверки допустимых аргументов функций.
- CompletionContributor: PermissibleArgumentCompletionContributor — автодополнение для аргументов функций.
- DocumentationProvider: NumpyDocumentationProvider — выделение документации из строк документации формата Numpy.
- TypeProvider: PredefinedTypeProvider — получение информации о типах из базы данных.
- TypeProvider: NumpyDocTypeProvider — получение информации о типах из строк документации Numpy.
- ApplicationConfigurable: TypeInformationConfigurable — панель редактирования пользовательской базы данных типизированных функций.
- pyModuleMembersProvider: NumpyModuleMembersProvider — поддержка динамических типов Numpy.

Кроме того, база данных для типизированных функций может быть использована для поддержки стандартной библиотеки Python в PyCharm вместо имеющихся property-файлов.