Python: модули, пакеты, документирование кода | Я.Шпора

Пользовательские модули и пакеты

Модуль в Python — файл с кодом. Этот код можно импортировать в другие программы или модули. Модули помогают организовывать и разделять код проектов на логические блоки.

Пользовательский модуль *mymodule.py…*

```
# mymodule.py

def print_hello(name):
    """Выводит приветствие."""
    print(f'Привет, {name}!')
```

…можно импортировать в другой модуль, например, *main.py*:

```
# main.py

import mymodule

mymodule.print_hello('Алиса')

# Вывод в терминал:
# Привет, Алиса!
```

Пакет — коллекция модулей, структурированная в виде каталога. Основная идея пакета — объединить связанные модули в одну группу. В корневом каталоге пакета обычно есть файл *__init__.py*, который может быть пустым или содержать код. Этот файл инициализирует пакет.

Пример структуры пакета:

```
folder/
```

```
|── mypackage/ # Это пакет.
|   ├── __init__.py
|   ├── module1.py # Это модуль в пакете.
|   └── module2.py # Ещё один модуль в пакете.
└── main.py
```

Предположим, что содержимое модуля *module1.py* такое:

```
# module1.py

def print_hello():
    print('Привет!')
```

Тогда использовать пакет *mypackage* в файле *main.py* можно так:

```
# main.py

# Взять из пакета mypackage модуль module1,
# а из модуля — функцию print_hello.
from mypackage.module1 import print_hello()


print_hello()

# Вывод в терминал:
# Привет!
```

Конструкция `if __name__ == '__main__'`

Такую конструкцию обычно добавляют в исполняемый скрипт. Это условный оператор, который читается как «если скрипт запущен напрямую, выполни этот код». Код, который находится в теле этого условного оператора, должен исполняться только в том случае, если файл запущен напрямую как программа.

```
def some_function():
    print('Эту функцию можно импортировать и использовать в других файлах
```

```
if __name__ == '__main__':
    print('Этот код выполняется только тогда, когда запускается файл с ним
    print('Если файл будет импортирован, то этот код не выполнится.')
    some_function()
```

Интроспекция

Интроспекция в программировании — способность программы во время выполнения исследовать типы и свойства объектов, которые содержатся в программе.

Функция `type()` возвращает тип объекта, переданного в качестве аргумента:

```
class Board:
    pass


game = Board()
print(type(game))
```

Функция `isinstance()` определяет принадлежность объекта к определённому классу:

```
...

print(isinstance(game, Board))
print(isinstance(game, str))

# Вывод в терминал:
# True
# False
```

Атрибут `__class__` позволяет уточнить класс объекта:

```
...

game = Board()
print(game.__class__)
```

```
# Вывод в терминал:
# <class 'gameparts.parts.Board'>
```

Функция `dir()` возвращает список атрибутов и методов, доступных для объекта:

```
...

game = Board()
print(dir(game))

# Вывод в терминал:
# ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq_
_', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '
__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne
__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__
', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Через словарь `__dict__`, доступный атрибуту `__class__`, можно получить атрибуты и методы, определённые только при создании объекта:

```
class Board:
    def make_step(self):
        pass


game = Board()

print(game.__class__.__dict__)

# Вывод в терминал:
# {'__module__': '__main__', 'make_step': <function Board.make_step at 0x(
# '__dict__': <attribute '__dict__' of 'Board' objects>,
# '__weakref__': <attribute '__weakref__' of 'Board' objects>, '__doc__':
```

Функция `getsource()` модуля `inspect` позволяет получить код объекта, например функции или метода:

```
from inspect import getsource

class Board:
    def make_step(self):
        pass


game = Board()

print(getsource(Board))

# Вывод в терминал:
# class Board:
#     def make_step(self):
#         pass
```

Функция `isfunction()` позволяет проверить, является ли переданный объект обычной функцией:

```
# Из модуля inspect импортировать функцию isfunction.
from inspect import isfunction

...

game = Board()

# make_step() - это функция?
print(isfunction(game.make_step))

# Вывод в терминал:
# False
```

Функцией `ismethod()` можно проверить, является ли переданный объект методом класса:

```
# Из модуля inspect импортировать функцию ismethod.
from inspect import ismethod
...
```

```
game = Board()

# make_step() - это метод?
print(ismethod(game.make_step))

#  Вывод в терминал: True
```

Документирование кода

Строки документации (докстринги, *dostring*) принято писать для функций, методов, классов и модулей.

Пример документированного кода:

```
"""Документация модуля. Описывает работу классов и функций.
Размещается в верхней части файла (начиная с первой строки).
"""


def tricky_func(self):
    """Описывает работу функции tricky_func."""
    ...


class Test:
    """Класс Test используется для демонстрации docstring.
    После docstring в классе нужна пустая строка."""

    def first(self):
        """Этот docstring описывает метод first() и
        демонстрирует перенос строки
        документации.
        """
        ...
```

К докстрингам можно обращаться программно через атрибут `__doc__` :

```
import math
```

```
# Что хорошего есть в библиотеке math?
print(math.__doc__)

# Вывод в терминал:
# This module provides access to the mathematical functions
# defined by the C standard.
```

Исключения

Исключения в Python — события, которые возникают во время выполнения программы и сигнализируют о том, что что-то пошло не так, как ожидалось.

В Python есть встроенные исключения, например, *IndexError*, *ValueError*, однако разработчик может создавать и собственные исключения.

Для этого нужно создать новый класс, унаследованный от встроенного класса `Exception` или другого встроенного исключения.

```
class FieldIndexError(IndexError):

    def __str__(self):
        return 'Введено значение за границами игрового поля'
```

Выбрасывание исключения производится с помощью ключевого слова `raise`:

```
raise FieldIndexError
```

Исключения выбрасываются в тех местах программы, где может возникнуть ошибка, например:

```
field_size = 5
index = int(input())
# Если введёный индекс больше 5, выбрасываем исключение.
if index > field_size:
    raise FieldIndexError
```

Обработка исключений

Обработка исключений происходит с использованием нескольких ключевых

слов: `try`, `except`, `else` и `finally`. Обрабатывать исключения необходимо, чтобы программа не останавливалась в момент возникновения исключения.

Пример:

```
try:
    # Блок кода, который может вызвать исключение.
    result = 10 / 0
except ZeroDivisionError as e:
    # Обработка исключения при делении на ноль.
    print(f'Ошибка: {e}')
else:
    # Необязательный блок кода, который выполняется, если исключение не в
    print('Операция выполнена успешно.')
finally:
    # Необязательный блок кода, который выполняется всегда.
    print('Программа завершила свою работу.')
```

Работа с файлами

Работа с файлами **всегда** состоит из трёх основных шагов:

1. Открыть файл.
2. Выполнить операции, например, прочитать файл или записать информацию в него.
3. Закрыть файл.

Чтобы открыть файл, используется функция `open()`.

```
f = open(<file>, <mode>)
```

- `file` — первый и обязательный аргумент. Он указывает на путь к файлу, который вы хотите открыть.
- `mode` — этот параметр определяет режим, в котором файл будет открыт.

Примеры режимов:

- `'r'` — чтение: по умолчанию открывает файл на чтение. Чтобы файл можно было прочитать, он должен существовать.
- `'w'` — запись: создаёт новый файл или перезаписывает существующий.

- `'a'` — добавление: добавляет данные в конец файла, при этом существующие данные не удаляет.
- `'b'` — двоичный режим: предназначен для чтения или записи двоичных файлов, например, изображений.

После выполнения операций с файлом, его нужно обязательно закрыть с помощью метода `close()`:

```
# Открыть файл example.txt для записи (аргумент 'w').
file = open('example.txt', 'w')
# Закрыть файл.
file.close()
```

Для добавления текста можно использовать метод `write()`:

```
file = open('example.txt', 'w')
# Записать в файл строку.
file.write('Зевну, укроюсь с головою,\nбудильник заведу на март.\n')
file.close()
```

> Если в терминале вы видите непонятные символы вместо букв, явно укажите кодировку, которая должна использоваться в файле. Это можно сделать при помощи опционального параметра `encoding`:
>
> ```
> file = open('example.txt', 'w', encoding='utf-8')
> ```

Для чтения из файла можно использовать метод `read()`. Опциональный параметр — `size`, который определяет количество символов, которые будут прочитаны из файла. Если `size` не указан или имеет отрицательное значение, то метод `read()` прочитает и вернёт содержимое файла целиком.

```
file = open('example.txt', 'r')
# Прочитать первые 11 символов из файла и сохранить их в переменную conte
content = file.read(12)
# Вывести на печать содержимое переменной content.
print(content)
```

```
# Закрыть файл.
file.close()
```

Контекстные менеджеры

Контекстный менеджер — конструкция, которая во время работы программы создаёт определённым образом настроенную среду («контекст»), где будет выполняться заданный код.

Чтобы выполнить код в среде контекстного менеджера, используется такой синтаксис:

```
with ContextManager as cm:
    # Код, который будет выполнен в контексте ContextManager.
```

У функции `open()` имеет встроенный контекстный менеджер:

```
with open('hello_bro.txt', 'w', encoding='utf-8') as f:
    f.write('Здравствуй, Стас!')
```

Полезные ресурсы

Правила оформления докстрингов

Параметры функции open() и режимы открытия файлов

Документация метода readline()

Документация методы readlines()

Среди бесконечных мыслей
шпаргалка ведёт мудрого к
совершенству.

Франсуа Карбон, неизвестный великий просветитель