Python: пространства имён, итераторы, генераторы, lambda, декораторы | Я.Шпора

Пространство имён и области видимости

Пространство имён — среда, в которой каждое имя (переменной, функции или класса) сопоставляется с уникальным объектом.

Пространства имён в Python:

- Встроенное пространство имён содержит имена всех встроенных объектов, которые доступны при работе в Python.

  Получить список всех имён объектов во встроенном пространстве:

  ```
  print(dir(__builtins__))
  ```

- Глобальное пространство содержит имена, определённые на уровне модуля.

  ```
  # Определить переменную value в глобальном пространстве имён.
  value = 'Я в глобальном пространстве имён!'


  def inner_function():
      value = 'А я нет!'
  ```

- Локальное пространство имён содержит имена, определённые внутри цикла, условия, функции, контекстного менеджера и так далее.

  ```
  def outer_function():
      # Это локальное пространство имен функции outer_function.
      print('Старт функции outer_function()')
      # Объявление вложенной функции
      def inner_function():
          # Это локальное пространство имен функции inner_function.
  ```

```
    print('Старт функции inner_function()')
    print('Завершение функции inner_function()')
```

Области видимости переменной

Области видимости определяют, какие пространства имён доступны для поиска нужного имени.

Области видимости в Python:

- Локальная область видимости. Если сослаться на какую-то переменную, то интерпретатор сначала будет искать её в локальном пространстве имён, которое относится к текущему функциональному блоку.

```
value = 'Глобальная value'


def outer_function():
    value = 'Локальная value из outer_function()'

    ...

    def inner_function():
        value = 'Локальная value из inner_function()'
        print(value)  # Обратились к переменной value.
        # Сперва ищем это имя в локальном пространстве имён,
        # внутри функции inner_function(). Нашли! Печатаем.

    inner_function()


outer_function()

# Вывод в терминал:
# Локальная value из inner_function()
```

- Внешняя (охватывающая) область видимости. Если запрошена переменная, которой нет в локальном пространстве имён, поиск продолжается в охватывающем пространстве.

```
value = 'Глобальная value'


def outer_function():
    value = 'Локальная value из outer_function()'

    ...

    def inner_function():
        # value = 'Локальная value из inner_function()'
        print(value)  # Обратились к переменной value.
        # Ищем это имя в локальном пространстве имён - нету!
        # Ищем в ближайшем внешнем пространстве имён,
        # в локальном пространстве функции outer_function(). Нашли!
Печатаем.

    inner_function()


outer_function()

# Вывод в терминал:
# value из outer_function()
```

- Глобальная область видимости. Если имя не найдено и в охватывающих пространствах имён, интерпретатор начнёт искать его в глобальной области видимости — в глобальном пространстве, которое обычно соответствует уровню модуля.

```
value = 'Глобальная value'


def outer_function():
    # value = 'Локальная value из outer_function()'

    ...
```

```
        def inner_function():
            # value = 'Локальная value из inner_function()'
            print(value)  # Обратились к переменной value.
            # Ищем это имя в локальном пространстве имён - нету!
            # Ищем в ближайшем внешнем пространстве имён - нету!
            # Ищем в глобальном пространстве имён. Нашли! Печатаем.

        inner_function()


    outer_function()

    # Вывод в терминал:
    # Глобальная value
```

- Встроенная область видимости. Если интерпретатор так и не нашёл переменную, он продолжит поиск во встроенном пространстве имён.

Если интерпретатор не найдёт имя и во встроенном пространстве имён, будет вызвано исключение `NameError`.

Инструкции `global` и `nonlocal`

В любом пространстве имён можно определить переменную как глобальную с помощью инструкции `global`:

```
my_var = 10


def my_function():
    # Описываем переменную как глобальную...
    global my_var
    # Теперь глобальную переменную можно обрабатывать прямо внутри функции
    # например - переопределить:
    my_var = 5
    print('Внутри функции:', my_var)
    # ...и снова переопределить:
    my_var += 95
    print('Внутри функции после изменений:', my_var)
```

```
my_function()
print('Снаружи функции:', my_var)

# Вывод в терминал:
# Внутри функции: 5
# Внутри функции после изменений: 100
# Снаружи функции: 100
```

Изменения такой переменной в локальном пространстве имён будут затрагивать глобальную переменную.

Для доступа к переменным из пространства имён охватывающей функции используется инструкция `nonlocal`.

```
def outer_function():
    value = 10

    def inner_function():
        nonlocal value
        value = 20

    inner_function()
    print(value)


outer_function()

# Вывод в терминал: 20
```

Словари пространств имён Python

Такие словари хранят информацию о глобальном и локальном пространствах имён.

Получить доступ к этим словарям можно через встроенные функции `globals()` и `locals()`.

```
global_value = 10
```

```
def any_function():
    local_value = 20
    local_text = 'Локальная строка'
    global global_value
    global_value = 100500  # Изменяем глобальную переменную.
    # Печатаем словарь с объектами локального пространства функции:
    print(f'Локальные переменные в функции any_function(): {locals()}')


any_function()

# Печатаем словарь с объектами глобального пространства программы:
print(f'Глобальные переменные программы: {globals()}')

# Вывод в терминал:
# Локальные переменные в функции any_function(): {'local_value': 20, 'l
ocal_text': 'Локальная строка'}
# Глобальные переменные программы: {'__name__': '__main__', ...}
```

Итераторы и генераторы

Итерируемым называют объект, который содержит элементы, и эти элементы можно перебрать. Каждый шаг перебора — это итерация.

В Python итерируемые объекты — это, например, списки, кортежи, строки.

Чтобы проверить, итерируется объект или нет, можно применить к нему функцию `iter()` . Для итерируемых объектов она вернёт объект итератора, а для неитерируемых — ошибку *object is not iterable.*

```
english_words = ['apple', 'banana', 'cherry', 'date', 'fig']
# Объект english_words итерируемый?
print(iter(english_words))

# Вывод в терминал:
# <list_iterator object at 0x0000027CDB3B9E10>

quantity = 5
```

```
# Объект quantity итерируемый?
print(iter(quantity))

# Вывод в терминал:
# TypeError: 'int' object is not iterable
```

Генератор — это подвид итератора: функция, которая генерирует значения.

Генератор не хранит элементы в оперативной памяти, а вычисляет их по заданному правилу; как следствие — генератор занимает в памяти минимум места.

```
# Создать функцию-генератор.
def short_sequence():
    num = 1
    while num < 5:
    # Сгенерировать значение через yield.
        yield num
        num += 1


# Здесь функция-генератор возвращает итератор.
step = short_sequence()

# Обратиться к методу __next__() итератора
# и получить первое значение последовательности.
print(step.__next__())

# Вывод в терминал:
# 1

# Ещё раз обратиться к методу __next__()
# и получить второе значение последовательности.
print(step.__next__())

# Вывод в терминал:
# 2
```

Когда значения исчерпаются, генератор выбросит исключение `StopIteration`.

Генераторное выражение — упрощённый синтаксис для создания генератора:

```
simple_generator = (digit for digit in range(2))+
print(type(simple_generator))
print(simple_generator.__next__())
print(simple_generator.__next__())

# Вывод в терминал:
# <class 'generator'>
# 0
# 1
```

lambda-функции

Безымянные, «анонимные» функции в Python, объявляются с помощью ключевого слова `lambda`.

Шаблон:

```
lambda параметры_через_запятую: выполняемое_функцией_выражение
```

Пример:

```
x = lambda x, y: x * y
print(x(5, 4))

# Лямбда-функция с параметрами x и y  возвращает произведение аргументо
в.
# Вывод в терминал: 20
```

Декораторы

Паттерны проектирования, предназначенные для изменения функциональности объектов без вмешательства в их код.

Структура декоратора:

```
def the_decorator(func):
    # Вложенная функция.
    def wrapper():
        ...
```

```
        result = func()  # Вызов функции, полученной в аргументах
        ...
        return result
    # Декоратор возвращает вложенную функцию.
    return wrapper
```

Применение декоратора:

```
def the_decorator(func):
    ...

def foo():
    ...

foo = the_decorator(foo)
```

Декорирование функции с помощью оператора `@` :

```
def the_decorator(func):
    ...

@the_decorator
def foo():
    ...
```

Декораторы для методов класса

Декоратор @classmethod

По умолчанию все методы в классе привязаны к экземпляру класса, а не к самому классу. С помощью декоратора `@classmethod` можно объявить метод, который привязан к классу, а не к его экземпляру.

```
class Robot:
    # Состояние батареи базовой станции:
    base_battery_status = 100

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

```
    # Декорируем и изменяем метод update_base_battery_status(),
    # чтобы менять значение атрибута не в объекте, а в классе:
    @classmethod
    def update_base_battery_status(cls, new_status):  # Указываем аргумен
        """
        Обновляет состояние батареи базовой станции.
        """
        # Присваиваем новое значение атрибуту класса.
        cls.base_battery_status = new_status

    def report(self):
        """
        Печатает в консоли состояние батареи базовой станции.
        """
        print(f'{self.name} reporting: Battery status is {self.base_batte

# Создаем двух роботов:
robot1 = Robot('R2-D2')
robot2 = Robot('C-3PO')

# Печатаем состояние батареи:
robot1.report()
robot2.report()

# Вывод в терминал:
# R2-D2 reporting: Battery status is 100%
# C-3PO reporting: Battery status is 100%

# Обновляем статус батареи в классе: обращаемся не к объекту, а к классу.
Robot.update_base_battery_status(80)

# Снова печатаем состояние батареи:
robot1.report()
robot2.report()

# Вывод в терминал:
```

```
# R2-D2 reporting: Battery status is 80%
# C-3PO reporting: Battery status is 80%
```

Декоратор @staticmethod

Статические методы в основном используются как вспомогательные функции и работают как обычные функции, обрабатывая данные, которые им переданы. Они описываются при помощи декоратора `@staticmethod`.

```
class Robot:
    ...
    @staticmethod
    def predict_battery_lifetime(current_capacity, charge_cycles):
        """
        Прогнозирует срок службы аккумулятора
        на основе текущей ёмкости и количества циклов зарядки.
        """
        # Пусть максимальная ёмкость нового аккумулятора будет равна 5000
        max_capacity = 5000
        return (current_capacity / max_capacity) * (1000 - charge_cycles)


# Вызов статического метода через имя класса:
battery_lifetime = Robot.predict_battery_lifetime(4000, 100)
print(
    'Прогноз срока службы аккумулятора: '
    f'осталось {battery_lifetime:.0f} циклов зарядки.'
)
# Прогноз срока службы аккумулятора: осталось 720 циклов зарядки.

# Создаём объект класса:
robot = Robot('R2-D2')
# Статический метод доступен и в объекте:
r2d2_battery_lifetime = robot.predict_battery_lifetime(3500, 150)
print(
    'Прогноз срока службы аккумулятора: '
    f'осталось {battery_lifetime:.0f} циклов зарядки.'
)
```

```
# Вывод в терминал:
# Прогноз срока службы аккумулятора: осталось 720 циклов зарядки.
```

Декоратор @property

Позволяет представить метод как свойство экземпляра класса.

```
class Robot:
    ...
    @property
    def identifier(self):
        """
        Вычисляет уникальный идентификатор робота на основе его имени.
        """
        # Преобразование имени в числовое представление:
        return sum(ord(char) for char in self.name)


# Создаём робота:
robot = Robot('R2-D2')
print(robot.identifier)

# Вывод в терминал:
# 295
```

Шпаргалка — мост, ведущий мудрого над безднами забвения к вершине знания.

Дге Тибетский