Рубежный контроль № 3: конспект по скриптовому языку

15 января 2025 г.

Родион Лавров, ИУ9-12Б

Язык программирования Python

1. Типизация и система типов языка

Тип данных	Неизменяемый	Описание	Пример
None	+	Имеет единственное значение	`None`
Bool	+	Булевы значения (`True`, `False`)	`True`, `False`
Bytes	+	Последовательность отдельных байтов	`bytes(10)`
bytearray	-	Последовательность отдельных байтов	`bytes(10)`
Int	+	Представление целых чисел	`-45`, `52`
Float	+	Числа с плавающей точкой	`14.676`, `5.43`
Complex	+	Комплексные числа	`1.23+0j`, `-1.23+4.5j`
Str	+	Последовательность кодовых точек Unicode	`"python"`, `"Hello World"`
Tuple	+	Хранение множества разнородных данных	`(32, 456)`
List	-	Хранение множества однородных элементов	`[32, 456]`

Тип данных	Неизменяемый	Описание	Пример
Range	+	Последовательность чисел, обычно используется для выполнения определенного количества циклов в циклах for	`list(range(5))`
Dict	-	Ассоциативный массив	`{"one": 1, "two": 2}`
Set	-	Хранение множества различных объектов с возможностью хэширования	`{'python', 'Hello World'}`
Frozenset	+	Неупорядоченная коллекция различных объектов с возможностью хэширования	`{'python', 'Hello World'}`

2. Основные управляющие конструкции

- Ветвление: `<if>::= if<condition>: <body> elif<condition>: <body> else: <body>`
- Цикл с условие: `<while-loop>::= while<condition>: <body>`
- Цикл с переменной: `<for-loop>::= for <var> in <iter-object>: <body>`
- Прерывние: `break`
- Переход к следующей итерации: `continue`
- Ничего не делает: `pass`
- Аналог switch-case: `<match-statement>::= match <var>: case <value>: <body>`
- Объявление функций: `<define-func>::= def <name>(<var>*): <body>`
- Обработка ошибок: `<try>::= try: <body> except <exception>: <body>`

3. Подмножество языка для функционального программирования

Способы обеспечить иммутабельность данных

Такие типы данных, как `Set`, `Dict`, `List`, `bytearray` поумолчанию являются мутабельными, в отличии от отстальных. Создать иммутабельный класс, можно различными способами. Вот несколько из них:

1) С помощью класса `property`, который позволяет задать определение,

получение и изменение полей класса

```
class Immutable():
   b = property(lambda s: "Hello World")

a = Immutable()
a.b = "mutated"  # AttributeError: property 'piece' of 'Immutable' object has no setter
```

2) С помощью метода `__setattr__`, который определяет каким образом будут изменяться поля класса

```
class Immutable():
    def __init__(self):
        self.b = 2

    def __setattr__(self, name, value):
        raise Exception(f"Cannot change value of {name}.")

a = Immutable()
a.b = 1    # Exception: Cannot change value of b.
```

3) С помощью метода `__slots__`, который не позволяет добавлять новые атрибутов после инициализации.

```
class Immutable:
    __slots__ = ('x')  # Ограничение атрибутов

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

a = Immutable(1)
a.z = 4  # AttributeError: 'Immutable' object has no attribute 'z'
```

Тем не менее, иммутабельность пользовательских классов в Python является лишь формальной, так как её можно обойти. Например в примере 2 это можно сделать следующем образом

```
a.__dict__["b"] = 1
```

Функции, как объекты 1-го класса в Python

Все данные в Python представлены объектами или отношениями между объектами. Поэтому функции могут быть присвоены другим переменным,

```
def func(string):
    return string.lower()

var = func
print(var("Hello World")) # hello world
```

переданы другим функциям в качестве аргуметов

```
def greet(func):
    greeting = func('Hi, I am a Python program')
    print(greeting)

greet(func) # hi, i am a python program
```

и возвращены из функции

```
def func2(func):
    return func

a = func2(func)("Hi, I am a Python program")
print(a) # hi, i am a python program
```

Функции высших порядков

Функция высшего порядка — это функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата. Реализация такой функции была приведена в примере выше. Рассмотрим встроенные функции высших порядков `map` и `filter`:

```
numbers = [5, 6, 7, 8, 9, 10]

new_numbers = map(lambda x: x ** 2, numbers)
print(list(new_numbers))   # [5, 8, 13, 21, 34, 55]

odd_numbers = filter(lambda x: x % 2 , numbers)
print(list(odd_numbers))   # [5, 7, 9]
```

4. Если выбранный язык — объектно-ориентированный, синтаксис определения простых классов, какие-то примечательные особенности ООП.

В Python класс определяется с помощью ключевого слова `class`, за которым следует имя и двоеточие.

Опередение

```
class Professor():
    pass
```

Жизненый цикл класса

При создании экземпляра класса, первым срабатывает метод `__new__`, после него – `__init__`. Метод `__new__` редко переопределяют, но он может быть полезен,

например, для реализации синглтонов. В `__init__` обычно инициализируются атрибуты экземпляра.

```
class Professor():
    def __new__(cls): # cls - ссылка на класс
        print("Calling __new__")
        return super().__new__(cls)

    def __init__(self, name, surname):
        print("Calling __init__")
        self.name = name
        self.surname = surname

prof = Professor("Vladimir", "Konovalov")
# Вывод:
# Calling __new__
# Calling __init__
```

Инкапсуляция

Python поддерживает базовые механизмы инкапсуляции, хотя строгих модификаторов доступа (как private или protected в других языках) нет. Атрибуты и методы могут быть:

- Публичными (доступны отовсюду),
- Защищёнными (начинаются с _, это соглашение о том, что они предназначены для внутреннего использования),
- Приватными (начинаются с __, создают обфускацию имени, что затрудняет доступ извне).

```
class Professor:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
        self._department = "IU9"
        self.__salary = 1000000000

    def teach(self):
        print(f"{self.surname} starts the lesson")

    def __estimate_rk(self):
        print(f"{self.surname} estimates RK by maximum points")

prof = Professor("Vladimir", "Konovalov")
print(prof.name)              # Публичный: работает
print(prof._department)       # Защищённый: работает, но использовать не рекомендуется
print(prof.__salary)          # AttributeError
```

Однако в Python даже к приватными атрибутам можно получить доступ.

```
print(prof._Professor__salary) # Вывод: 1000000000
```

Наследование

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

    def introduce(self):
        print(f"My name is {self.name} {self.surname}.")

class Professor(Person):
    def __init__(self, name, surname, university):
        super().__init__(name, surname)
        self.university = university

    def introduce(self):
        print(f"My name is Prof. {self.surname}, I work in {self.university}.")

prof = Professor("Vladimir", "Konovalov", "BMSTU")
prof.introduce() # My name is Prof. Konovalov, I work in BMSTU.
```

Функция `super()` используется для вызова методов родительского класса. Обычно применяется в переопределённых методах для вызова реализации родителя.

Полиморфизм

```
class Person:
    def introduce(self):
        print("I am a person.")

class Professor(Person):
    def introduce(self):
        print("I am a professor.")

class Student(Person):
    def introduce(self):
        print("I am a student.")

people = [Person(), Professor(), Student()]

for person in people:
    person.introduce()
```

```
# Вывод:
# I am a person.
# I am a professor.
# I am a student.
```

Абстракция

```
from abc import ABC, abstractmethod

class Professor(ABC):
    @abstractmethod
    def evaluate(self):
        pass

class CSProfessor(Professor):
    def evaluate(self):
        print("Проверка лабораторных работ.")

class MathProfessor(Professor):
    def evaluate(self):
        print("Проверка контрольных по математике.")

CSProfessor().evaluate()
MathProfessor().evaluate()
```

Примечательные особенности Python ООП

1) Динамическое добавление атрибутов
   Атрибуты могут быть добавлены или изменены после создания объекта.

```
prof.specialization = "Scheme"
print(prof.specialization)
```

2) Декораторы классов и методов Декораторы позволяют изменять или добавлять функциональность к классам и методам.

```
def add_repr(cls):
    cls.__repr__ = lambda self: f"{cls.__name__}({self.__dict__})"
    return cls

@add_repr
class MyClass:
    def __init__(self, x):
        self.x = x
```

```
obj = MyClass(42)
print(obj)  # MyClass({'x': 42})
```

3) Утиная типизация (Duck Typing) Python использует подход, при котором неважно, к какому классу принадлежит объект. Главное — какие методы и свойства он поддерживает.

```
class Duck:
    def quack(self):
        print("Quack!")

class Person:
    def quack(self):
        print("I can quack too!")

Duck().quack()    # Quack!
Person().quack()  # I can quack too!
```

5. Важнейшие функции для работы с потоками ввода/вывода, строками, регулярными выражениями.

Работа с потоками ввода/вывода

Python предоставляет удобный интерфейс для работы с файлами и потоками. Основные функции:

- Открытие файлов: `open(file, mode)`
- Режимы: `r` (чтение), `w` (запись), `a` (добавление), `b` (двоичный режим).
- Чтение данных:
- `read(size)` - чтение всего содержимого или указанного количества символов.
- `readline()` - чтение одной строки.
- `readlines()` - чтение всех строк файла.
- Запись данных: `write(data)` — запись строки или байтов.
- Закрытие файла: `close()` или использование контекстного менеджера `with`.

Пример: чтение и запись файла

```
with open('file.txt', 'r') as file:
    print(file.read())

with open('file.txt', 'w') as file:
    file.write("Hello, world!\nThis is a test file.")
```

Работа со строками

Python предоставляет обширный набор методов для обработки строк:

- Модификация строк: `lower()`, `upper()`, `capitalize()`, `title()`, `strip()`, `replace(old, new)`.
- Проверка содержимого: `startswith(prefix)`, `endswith(suffix)`, `isalnum()`, `isalpha()`, `isdigit()`.
- Разделение и объединение: `split(delimiter)`, `join(iterable)`.
- Форматирование строк:
  - Старый стиль: `"%s %d" % ("Hello", 123)`.
  - Новый стиль: `"{} {}".format("Hello", 123)`.
  - f-строки: `f"{variable}"`.

Пример: работа со строками

```
text = "Python is Awesome!"
clean_text = text.strip().lower()
print(clean_text)  # "python is awesome!"
```

Регулярные выражения

Регулярные выражения используются для поиска и обработки текста с помощью модуля `re`. Основные функции:

- `match(pattern, string)` — проверка, начинается ли строка с шаблона.
- `search(pattern, string)` — поиск первого совпадения в строке.
- `findall(pattern, string)` — поиск всех совпадений.
- `sub(pattern, repl, string)` — замена по шаблону.
- `compile(pattern)` — компиляция шаблона для повторного использования.

Пример регулярных выражений:

```
import re

text = "Email: example@mail.com, phone: +123456789"
email = re.search(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)
print(email.group())
cleaned_text = re.sub(r'\+\d+', '[hidden]', text)
print(cleaned_text)
```