НИИТ

Лекция №2

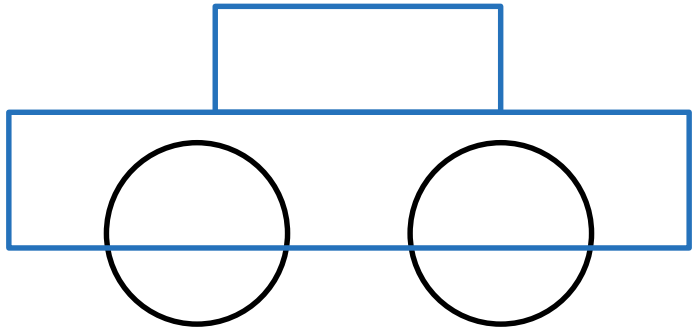
Функции и объекты

- Функции и функциональный подход
- Введение в ООП
- Переменные, объекты и динамическая типизация
- Сборщик мусора

python

Функции и функциональный подход

Задача поставлена следующим образом: написать алгоритм отрисовки автомобиля. ОС предлагает ограниченный API (application programming interface): draw_pixel(x, y), где x, y – координаты закрашиваемого пикселя.



Функции и функциональный подход

Функция - это блок организованного, многократно используемого кода, который используется для выполнения конкретного задания. Функции обеспечивают лучшую модульность приложения и значительно повышают уровень повторного использования кода. **Функция определяется набором принимаемых аргументов и типом возвращаемого значения.**

```
# def - baseword для определения
# hello_world - название
def hello_world ():
    print 'Привет, Мир!' # блок, принадлежащий функции, print - тоже функция
# Конец функции
hello_world() # вызов функции
hello_world() # ещё один вызов функции


# a и b - параметры функций
>>> def test(a, b):
...     print "output: {}".format(a + b)
...
>>> test(1, 5) # 1 и 5 - это уже аргументы функций
output: 6
>>> test(2, 5)
output: 7
```

Функции и функциональный подход

```
# есть аргументы со значением
# по умолчанию
def test(a, b=3):
    print a, b
test(1, 4)
test(1)
```

```
# именованные аргументы можно передавать
# в любом порядке
def test(a, b=3, c=5):
    print a, b, c

test(1, c=10, b=8)
```

```
# с помощью * в списке аргументов можно передать любое количество аргументов
def test(a, *args):
    print "a: {}".format(a)
    for arg in args:
        print arg
test(1, 2)
test(1, 2, 3, 4, 5)
my_list = [2, 3, 4]
test(1, *my_list)
```

```
# с помощью ** в списке аргументов можно передать именованные аргументы
def test(a, **kwargs):
    print "a: {}".format(a)
    print "b: {}".format(kwargs['b'])
test(1, b=2)
test(1, b=2, c=3)
my_dict = {'b': 2}
test(1, **my_dict)
```

Функции и функциональный подход

```
# есть аргументы со значением
# по умолчанию
def test(a, b=3):
    print a, b
test(1, 4)
test(1)
```

```
# именованные аргументы можно передавать
# в любом порядке
def test(a, b=3, c=5):
    print a, b, c

test(1, c=10, b=8)
```

```
# с помощью * в списке аргументов можно передать любое количество аргументов
def test(a, *args):
    print "a: {}".format(a)
    for arg in args:
        print arg
test(1, 2)
test(1, 2, 3, 4, 5)
my_list = [2, 3, 4]
test(1, *my_list)
```

```
# с помощью ** в списке аргументов можно передать именованные аргументы
def test(a, **kwargs):
    print "a: {}".format(a)
    print "b: {}".format(kwargs['b'])
test(1, b=2)
test(1, b=2, c=3)
my_dict = {'b': 2}
test(1, **my_dict)
```

Функции и функциональный подход

```
# return - тут все понятно
def test():
    return 5
x = test()
print(x)


# pass - это заглушка
def test():
    pass
x = test()
print(x)


# lambda - анонимная функция
test = lambda x: print(x)
test(1)


# явное указание типа аргумента
# ничем не помогает, но становится понятней
def test(x: int):
    return x
```

Объектно-ориентированное программирование (ООП)

Задача посложнее: написать алгоритм работы приложения для заказа такси.

Принципы ООП:

1. Абстрагирование (создаем модель реального объекта, исключая его характеристики, не требуемые для решения поставленной задачи).
2. Инкапсуляция (модель – это «черный ящик»: пользователям известны допустимые входные данные, известно, что ожидать на выходе, но устройство и реализация их волновать не должны).
3. Наследование (если модели имеют одинаковые функциональности, выносим эти функциональности в общую родительскую модель).
4. Полиморфизм (алгоритм, умеющий работать с родительской моделью, будет работать и с дочерними).

и многие другие (см. SOLID – «single responsibility, open-closed, Liskov substitution, interface segregation и dependency inversion»).

Объектно-ориентированное программирование (ООП)

Как происходит проектирование программы с использованием принципов ООП?

1. Анализируем предметную область, выделяя объекты, т.е. сущности с определенным состоянием и поведением. Используя принцип абстрагирования, определяем важные для решения задачи аспекты объекта, выделяя их в классы, идем от частного к общему, по примеру определяем формулу.
2. По принципу инкапсуляции делаем классы максимально независимыми и изолированными друг от друга, ослабляем зависимости между классами (правило «разделяй и властвуй»).

Объектно-ориентированное программирование (ООП)

3. Из родственных классов, чтоб избежать дублирования в их содержании и поведении, организуем иерархию, применяя принцип наследования, например, для двух классов: автомобиль и мотоцикл, - вместо указания в описании каждого марки, модели, максимальной скорости переносим эти общие характеристики в третий класс – транспорт, объявляя его родительским для классов автомобиля и мотоцикла. Последние теперь будут содержать только исключительно им присущие характеристики.
4. Ну и наконец, для того, чтоб алгоритмы, работающие с данными классами, минимально зависели от конкретики этих классов, применяем принцип полиморфизма, позволяя программе уже в момент исполнения переходить от общего к частному. Т.е. алгоритм, написанный для работы с классом транспорт, сможет работать и с классами автомобиль и мотоцикл, и при этом, если необходимо, задействовать специфические функциональности этих классов.

Объектно-ориентированное программирование (ООП)

Рассмотрим, как это работает, на примере следующей задачи:

Надо написать алгоритм приготовления пиццы. Основные шаги приготовления следующие:

- подготовить тесто,
- добавить ингредиенты,
- испечь.

За порядок шагов и за сами шаги должны отвечать разные объекты – принцип единственной обязанности.

Пицца может быть разная: грибная, мясная, овощная. Тесто может готовиться одинаково, печься может одинаково, а вот ингредиенты могут добавляться и даже готовиться по-разному.

Объектно-ориентированное программирование (ООП)

Что такое класс или тип? Проведем аналогию с реальным миром. Если мы возьмем конкретный автомобиль, то это объект, но не класс. А вот общее представление об автомобилях, их назначении – это класс. Ему принадлежат все реальные объекты автомобилей, какими бы они ни были. Класс автомобилей дает общую характеристику всем автомобилям в мире, он их обобщает.

Рассмотрим целые числа в Python. Тип int – это класс целых чисел. Числа 5, 20344, -10 и т. д. – это конкретные объекты этого класса.

Любая программа работает с данными. Данные в языке Python представлены в форме объектов - встроенных, предоставляемых языком Python, или объектов, которые мы создаем с помощью других инструментов. По сути, объекты – это области памяти со значениями и ассоциированными с ними наборами операций (которые и отождествляются с типом объекта).

Переменные, объекты и динамическая типизация

Между программами и объектами выстроена целая иерархия (это справедливо как в Python, так и в других языках программирования):

- Программы делятся на модули.
- Модули содержат инструкции.
- Инструкции состоят из выражений.
- Выражения создают и обрабатывают объекты.

Переменные, объекты и динамическая типизация

Любой объект имеет тип. Поскольку Python использует динамическую типизацию, заранее объявлять тип переменной, ссылающейся на объект, не нужно. Механизм создания и определения переменных выглядит следующим образом:

- Переменная объявляется, как только ей присваивается некоторое значение
- Переменная не имеет типа, тип имеют объекты. А переменная - это просто ссылка, указывающая на конкретный объект в данный момент.
- Когда переменная используется, ее имя замещается объектом, на который она указывает.
- Неинициализированную переменную использовать нельзя.

Переменные, объекты и динамическая типизация

Если ввести такую инструкцию:

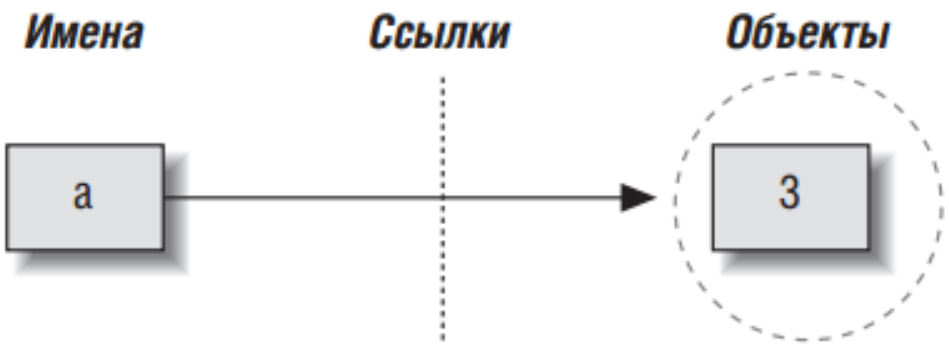
```
>>> a = 3
```

интерпретатор Python выполнит эту инструкцию в три этапа (по крайней мере, концептуально):

- Создается объект, представляющий число 3.
- Создается переменная a, если она еще отсутствует.
- В переменную a записывается ссылка (адрес в памяти) на вновь созданный объект, представляющий число 3.

Переменные, объекты и динамическая типизация

Переменные и объекты хранятся в разных частях памяти и связаны между собой ссылкой (ссылка на рисунке показана в виде стрелки). Переменные всегда ссылаются на объекты и никогда – на другие переменные, но крупные объекты могут ссылаться на другие объекты (например, объект списка содержит ссылки на объекты, которые включены в список). Когда бы ни использовалась переменная (то есть ссылка), интерпретатор Python автоматически переходит по ссылке от переменной к объекту.



Переменные, объекты и динамическая типизация

С точки зрения определений все это выглядит так:

Переменные – это записи в системной таблице, где предусмотрено место для хранения ссылок на объекты.
Объекты – это области памяти с объемом, достаточным для представления значений этих объектов.
Ссылки – это автоматически разыменовываемые указатели на объекты.

Python, создавая новое значение, всегда выделяет новый объект, то есть выделяет участок памяти. Но иногда Python хранит в кэше некоторые значения, оптимизируя все. Так, любая 1 или 97, используемая в программе - это один и тот же объект. А 99998 в одном месте и то же число в другом - не одно и то же.

Переменные, объекты и динамическая типизация

Нужно понимать, что внутри у стандартного Python сишный код. Так, например, в базовом варианте выглядит любой объект:

```
typedef struct _object
{
    Py_ssize_t ob_refcnt; // счетчик ссылок
    struct _typeobject *ob_type; // тип объекта
} PyObject;
```

Счётчик ссылок — это число, показывающее, сколько раз другие объекты ссылаются на данный объект, или сколько переменных хранят этот объект.

```
>>> a = b = c = object() # счетчик объекта увеличивается на три
```

Когда счетчик становится равен 0, сборщик мусора CAM удаляет объект и высвобождает память.

Важно помнить - в Python все и всегда передается по ссылке. Но неизменяемые (immutable) типы ведут себя так, как будто передаются по значению.

Переменные, объекты и динамическая типизация

Нужно понимать, что внутри у стандартного Python сишный код. Так, например, в базовом варианте выглядит любой объект:

```
typedef struct _object
{
    Py_ssize_t ob_refcnt; // счетчик ссылок
    struct _typeobject *ob_type; // тип объекта
} PyObject;
```

Счётчик ссылок — это число, показывающее, сколько раз другие объекты ссылаются на данный объект, или сколько переменных хранят этот объект.

```
>>> a = b = c = object() # счетчик объекта увеличивается на три
```

Когда счетчик становится равен 0, сборщик мусора CAM удаляет объект и высвобождает память.

Важно помнить - в Python все и всегда передается по ссылке. Но неизменяемые (immutable) типы ведут себя так, как будто передаются по значению.

НИИТ

Сборщик мусора

- Сборщик мусора имеет три поколения, при создании объекта он попадает в нулевое поколение. У каждого поколения есть счётчик (счетчик количества объектов) и порог. Работает эта пара так:
  - При добавлении объекта в поколение счётчик увеличивается.
  - При выбывании из поколения счётчик уменьшается.
  - Когда счётчик превысит пороговое значение — по всем объектам из поколения пройдётся сборщик мусора. Кого найдёт — удалит.
- Все выжившие в поколении объекты перемещаются в следующее (из нулевого в первое, из первого во второе). Из второго поколения объекты никуда не попадают и остаются там навечно.
- Перемещённые в следующее поколение объекты меняют соответствующий счетчик, и операция может повториться уже для следующего поколения.
- Счётчик текущего поколения сбрасывается.

Практика

1. Написать и вызвать функцию, принимающую два числа и выводящую на экран большее из двух.
2. Написать и вызвать функцию, принимающую два числа и возвращающую большее из двух.
3. * Провести объектно-ориентированный анализ задачи и определить объекты, которые потребуется использовать (подсказка: необходимо инкапсулировать изменяемые аспекты поведения объектов):
   - создается компьютерная игра, поддерживающая одновременное управление несколькими персонажами;
   - каждый персонаж умеет бегать, стрелять и собирать предметы;
   - некоторые персонажи умеют летать;
   - каждый тип действий сопровождается определенной анимацией;
   - собираемые предметы могут менять характеристики действий и соответствующую анимацию (например, вдвое ускорять или, наоборот, замедлять бег персонажа).