Лабораторная работа №2. Введение в объектно-ориентированное программирование.

Цель работы:

Ознакомить обучающегося с основными принципами объектно-ориентированного программирования на языке Python.

Введение

Функции на языке Python

Функции в программирование помогают группировать и обобщать программный код, который может позднее использоваться произвольное число раз. Она является законченной подпрограммой, поэтому у нее есть свои "ввод" и "вывод" — параметры (аргументы) и возвращаемое значение.

С точки зрения внешней программы функция — это "черный ящик". Функция определяет собственную (локальную) область видимости, куда входят входные параметры, а, также, те переменные, которые объявляются непосредственно в теле самой функции.

Главное, что должно быть можно сделать с функцией — это возможность ее вызвать.

Функции — это средство проектирования, которое позволяет осуществить декомпозицию программы на достаточно простые и легко управляемые части. Значительно проще написать решение маленьких задач по отдельности, чем реализовать весь процесс целиком. Устранение избыточности программного кода улучшает сопровождаемость кода — если что-то необходимо будет исправить, достаточно будет внести изменения всего в одном месте, а не во многих.

В Python функция записывается следующим образом: имя_функции (параметры_через_запятую). Чтобы вызвать функцию, после ее имени нужно указать круглые скобки и поместить внутрь параметры, отделив каждый из них запятой. Для создания функций в Python выберите ее имя, определите параметры, укажите, что функция должна делать и какое значение возвращать.

```
def имя_функции(параметры) :
    определениие _функции
```

Ключевое слово def сообщает Python, что вы определяете функцию. После def вы указываете имя функции; оно должно отвечать тем же правилам, что и имена переменных.

Простейшая функция может иметь вид:

```
def greet():
      print("Hello world!")
```

Такая функция, конечно, не несет никакой практической пользы и служит лишь для иллюстрации задания функции на языке Python.

Цель создания функции, как правило, заключается в том, чтобы вынести участок кода, который выполняет определенную задачу, в отдельный объект. Это позволяет использовать этот код многократно, не создавая его заново в программе.

Как правило, функция должна выполнять какие-то действия с входящими значениями и на выходе выдавать результат.

При работе с функциями различают два понятия: параметры и аргументы.

Параметры функции определяются именами, которые обозначаются при определении функции. Аргументами функции являются значения которые передаются функции при ее вызове. Параметры определяют, какой тип переменных может принимать функция.

Параметром функции называют именованную сущность в описании функции (или метода), указывающий аргумент (или, в некоторых случаях, аргументы), который функция может принять.

A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

Parameters are defined by the names that appear in a function definition, whereas arguments are the values actually passed to a function when calling it. Parameters define what kind of arguments a function can accept. For example, given the function definition:

Параметры бывают обязательные и необязательные.

Обязательные:

```
def f(a, b):
    pass
```

Необязательные (со значением по умолчанию):

```
def f(a=None):
    pass
```

В этом случае a - передавать необязательно.

Аргументы бывают позиционные и ключевые.

```
def summ(a, b):
    print(a + b)
```

Позиционные:

```
summ(1, 2)
```

Ключевые:

```
summ(a=1, b=2)
```

Независимо от того как параметры созданы, при вызове функции им можно передавать значения и как ключевые и как позиционные аргументы. При этом обязательные параметры надо передать в любом случае, любым способом (позиционными или ключевыми), а необязательные можно передавать, можно нет. Если передавать, то тоже любым способом.

Функция может возвращать результат. Для этого в функции используется оператор return, после которого указывается возвращаемое значение:

```
def имя_функции ([параметры]):
    инструкции
```

```
    return возвращаемое_значение
```

Математическая функция f(x) = x * 2 в Python будет выглядеть вот так:

```
def f(x):
    return x * 2
```

Затем это результат функции можно присвоить переменной или использовать как обычное значение:

```
double_x = f(2)  # получаем результат функции f в переменную double_x
print(double_x)          # 4

# можно напрямую передать результат функции f
print(f(2))    # 4
```

Оператор return не только возвращает значение, но и производит выход из функции. Поэтому он должен определяться после остальных инструкций. Например:

```
def f(x):
    return x * 2
    print("Some message")
```

С точки зрения синтаксиса данная функция корректна, однако ее инструкция print("Some message") не имеет смысла - она никогда не выполнится, так как до ее выполнения оператор return возвратит значение и произведет выход из функции.

Позиционные и именованные аргументы

Лямбда функции в Python

Лямбда-функции в Python являются анонимными. Это означает, что функция безымянна. Как известно, ключевое слов def используется в Python для определения обычной функции. В свою очередь, ключевое слово lambda используется для определения анонимной функции.

Лямбда-функция имеет следующий синтаксис:

```
lambda аргументы: выражение
```

Лямбда-функции могут иметь любое количество аргументов, но у каждой может быть только одно выражение. Выражение вычисляется и возвращается. Эти функции могут быть использованы везде, где требуется объект-функция.

```
double = lambda x: x*2
print(double(5)) #10
```

В вышеуказанном коде lambda x: x*2 — это лямбда-функция. Здесь x — это аргумент, а x*2 — это выражение, которое вычисляется и возвращается.

Эта функция безымянная. Она возвращает функциональный объект с идентификатором double.

Инструкция:

```
double = lambda x: x*2
```

Эквивалентна:

```
def double(x):
    return x * 2
```

Лямбда-функции часто используются, когда ненадолго необходима безымянная функция.

В Python они часто используются их как аргумент функции высшего порядка (функции, которая принимает другие функции в качестве аргументов). Лямбда-функции используют вместе с такими встроенными функциями как filter(), map(),reduce() и др.

Вот пример использования функции filter() для отбора четных чисел из списка:

```
my_list = [1, 3, 4, 6, 10, 11, 15, 12, 14]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)  #[4, 6, 10, 12, 14]
```

Аннотации в Python

Аннотации нужны для того чтобы повысить информативность исходного кода, и иметь возможность с помощью сторонних инструментов производить его анализ. Одной из наиболее востребованных, в этом смысле, тем является контроль типов переменных. Несмотря на то, что Python – это язык с динамической типизацией, иногда возникает необходимость в контроле типов.

Согласно PEP 3107 могут быть следующие варианты использования аннотаций:

- проверка типов;
- расширение функционала IDE в части предоставления информации об ожидаемых типах аргументов и типе возвращаемого значения у функций;
- перегрузка функций и работа с дженериками;
- взаимодействие с другими языками;
- использование в предикатных логических функциях;
- маппинг запросов в базах данных;
- маршалинг параметров в RPC (удаленный вызов процедур).

В функциях возмжно аннотировать аргументы и возвращаемое значение. Выглядеть это может так.

```
def repeater(s: str, n: int) -> str:
   return s * n
```

Аннотация для аргумента определяется через двоеточие после его имени.

```
имя_аргумента: аннотация
```

Аннотация, определяющая тип возвращаемого функцией значения, указывается после ее имени с использованием символов ->

```
def имя_функции() -> тип
```

Для лямбд аннотации не поддерживаются.

Модули в языке Python

Python — достаточно мощный ЯП с поддержкой модульного программирования. Модульное программирование представляет собой процесс разделения одной комплексной задачи программирования на несколько маленьких и более управляемых подзадач/модулей. Модули напоминают кирпичики Лего, которые образуют большую задачу, если собрать их вместе.

Модульность обладает множеством преимуществ при написании кода:

- Возможность повторного использования
- Поддерживаемость
- Простота

Функции, модули и пакеты обеспечивают модуляризацию кода в Python.

Модуль — это сценарий .py, который можно вызвать в другом сценарии .py. Модуль — это файл, содержащий определения и операторы Python, который участвует в реализации набора функций. Название модуля эквивалентно названию файла с расширением .py. Для импорта одних модулей из других используется команда import. Импортируем модуль math.

```
# import the library
import math
#Using it for taking the log
math.log(10)
2.302585092994046
```

Встроенные модули в Python

В Python имеется бесчисленное количество встроенных модулей и пакетов практически на любой случай. Полный список можно посмотреть здесь.

При изучении модулей в Python пригодятся две функции — dir и help.

Встроенная функция dir() используется для обнаружения функций, реализованных в каждом модуле. Она возвращает отсортированный список строк:

```
print(dir(math))
```

Функция help внутри интерпретатора Python предоставляет информацию об определенной функции в модуле:

```
help(math.factorial)
```

Packages

Пакеты — это коллекция модулей, собранных вместе. Базовые пакеты машинного обучения — Numpy и Scipy — состоят из коллекции сотен модулей. Ниже приведен неполный список подпакетов, доступных в SciPy.

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

Конструкция *if __ name__ == __"main__"*

Часто, изучая программы, написанные на языке Python, можно встретить конструкцию *if __ name__ == __"main__"*. При импорте модуля его код выполняется целиком. Соответственно, if name == "__main__" это просто способ, позволяющий избежать побочных эффектов от этого (в случае если модуль содержит код, который имеет смысл выполнять только при запуске скрипта напрямую, например, набор тестов).

Такая проверка придумана людьми, привыкшими программировать на С-подобных процедурных языках, где исходный код организован в виде программы с одной точкой входа.

Таким образом, если вы пишете программу, которая не предполагает ее использование как модуля, необходимость в использовании конструкции *if __ name__ == __«main__»* пропадает и этого можно не делать.

Объектно-ориентированное программирование на языке Python

Объектно-ориентированное программирование (ООП) - это парадигма программирования, которая ориентирована на понимание мира в терминах объектов. Объект - это сущность, которая содержит состояние и поведение. Состояние объекта - это его свойства, а поведение - это то, что он может делать.

В Python объекты создаются с помощью классов. Класс - это шаблон, определяющий состояние и поведение объектов, которые могут быть созданы на его основе. Каждый объект, созданный на основе класса, называется экземпляром этого класса.

В Python классы определяются с помощью ключевого слова class, а методы - это функции, определенные внутри класса. Обычно методы используются для работы со свойствами объекта.

Независимо от парадигмы программы используют для решения проблем одну последовательность действий:

- *Ввод данных*: данные считываются из какого-либо источника, которым может быть хранилище данных, например файловая система или база данных.
- *Обработка*: данные интерпретируются и, возможно, изменяются для подготовки к отображению.
- *Вывод данных*: данные представляются таким образом, чтобы их могли прочитать и использовать физический пользователь или система.

Класс — это шаблон объекта.

Следует отметить, что ООП не лучше и не хуже любой другой парадигмы. У каждой из них есть свои преимущества и недостатки. ООП имеет несколько полезных преимуществ, например:

- *Инкапсуляция данных*. Инкапсуляция данных заключается в скрытии данных от остальной части системы и разрешении доступа к отдельным ее частям. Причина в том, что данные содержат состояние, которое может состоять из одной или нескольких переменных. Если эти переменные нужно изменить одновременно, их необходимо защитить и разрешить доступ только через открытые методы, чтобы изменения происходили предсказуемо. ООП включает такие механизмы, как уровни доступа, когда данные в объекте доступны только самому объекту и могут быть сделаны общедоступными.
- *Простота*. Создание крупных систем — это сложная задача, которая требует решения множества проблем. Способность разбивать сложную задачу на небольшие проблемы, объекты, позволяет упростить общую задачу.
- *Простое изменение*. Полагаясь на объекты и моделируя систему с их помощью, проще отследить, какие части системы требуют изменения. Например, может потребоваться исправление ошибки или добавление новой функции.
- *Удобство обслуживания*. Поддерживать код непросто, а со временем становится все труднее. Помимо прочего это требует дисциплины в виде хорошего именования и четкой, согласованной архитектуры. Использование объектов упрощает поиск той части кода, которая требует обслуживания.
- *Многократное использование*. Определение объекта можно использовать много раз во многих частях системы и даже в других системах. Используя код повторно, вы экономите время и деньги, так как вам нужно писать меньше кода и вы быстрее достигаете цели.

Давайте создадим класс, который описывает человека:

```
class Person:
    pass
```

Для объявления класса Person мы использовали ключевое слово class. Из классов мы получаем экземпляры, созданные по подобию этого класса.

Объект

Объект — это экземпляр класса. Объявленный класс — это лишь описание объекта: ему не выделяется память.

Например, экземпляра класса Person будет выглядеть так:

```
# obj -- экземпляр класса Person
obj = Person()
```

Теперь разберемся, как написать класс и его объекты.

```
# Создаем класс и его объекты
class Person:

    # атрибуты класса
    species = "человек"

    # атрибуты экземпляра
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
# создаем экземпляра класса
vasy = Person("Василий", 22)
ivan = Person("Иван", 32)
```

Во многих языках программирования существует понятие конструктора — специальной функции, которая вызывается только при первом создании объекта. Конструктор будет вызываться только один раз. В этом методе вы создаете атрибуты, которые должен иметь объект. Кроме того, вы присваиваете созданным атрибутам начальные значения.

В Python конструктор называется __init__(). Конструктору также необходимо передать специальный параметр — self. Параметр self ссылается на экземпляр объекта. Присваивание этого ключевого слова означает, что атрибут попадает в экземпляр объекта. Если не добавить атрибут к self, он будет рассматриваться как временная переменная, которая не будет существовать после выполнения __init__().

Параметр self также необходимо передать любым методам, которые должны ссылаться на что-либо в экземпляре объекта.

Исключения

Исключительные ситуации или исключения (exceptions) – это ошибки, обнаруженные при исполнении. Например, к чему приведет попытка чтения несуществующего файла? Или если файл был случайно удален пока программа работала? Такие ситуации обрабатываются при помощи исключений.

Если же Python не может понять, как обойти сложившуюся ситуацию, то ему не остается ничего кроме как поднять руки и сообщить, что обнаружил ошибку. В общем, исключения необходимы, чтобы сообщать программисту об ошибках.

Простейший пример исключения - деление на ноль:

```
>>> 100 / 0
Traceback (most recent call last):
  File "", line 1, in
    100 / 0
ZeroDivisionError: division by zero
```

В данном случае интерпретатор сообщил нам об исключении ZeroDivisionError – делении на ноль.

Traceback

В большой программе исключения часто возникают внутри. Чтобы упростить программисту понимание ошибки и причины такого поведения Python предлагает Traceback или в сленге – трэйс. Каждое исключение содержит краткую информацию, но при этом полную, информацию о месте появления ошибки. По трэйсу найти и исправить ошибку становится проще.

Рассмотрим такой пример:

```
Traceback (most recent call last):
  File "/home/username/Develop/test/app.py", line 862, in _handle
    return route.call(**args)
  File "/home/username/Develop/test/app.py", line 1729, in wrapper
    rv = callback(*a, **ka)
  File "/home/username/Develop/test/__init__.py", line 76, in wrapper
    body = callback(*args, **kwargs)
  File "/home/username/Develop/test/my_app.py", line 16, in index
    raise Exception('test exception')
```

В данном примере четко видно, какой путь исполнения у программы. Смотрим снизу вверх и по шагам понимаем, как же мы докатились до такого исключения.

Задание к лабораторной работе

Реализовать класс “Матрица”, в котором переопределены функции сложения, умножения, вычитания таким образом, чтобы осуществлять операции с матрицами, предусмотреть возможность заполнения матрицы случайными числами (при создание объекта класса Матрица), функции транспонирования матрицы и вывода на экран, а также реализовать генерацию исключений в тех случаях, когда невозможно произвести требуемую операцию над матрицей (необходимо создать соответствующие исключения, например, MatrixDimensionError - несоответствие размеров матриц).

В качестве основы для реализации матрицы используйте следующий фрагмент кода:

```
class Matrix:
    def __init__(self, rows, cols, fill_random=False, min_val=0,
max_val=10):
        """Конструктор класса Матрица. Заполняет матрицу нулями или
случайными числами, если это указано."""
        pass

    def __str__(self):
        """Переопределение функции вывода на экран."""
        pass

    def __add__(self, other):
        """Переопределение операции сложения матриц."""
        pass

    def __sub__(self, other):
        """Переопределение операции вычитания матриц."""
        pass

    def __mul__(self, other):
        """Переопределение операции умножения матриц."""
        pass

    def transpose(self):
        """Функция транспонирования матрицы."""
       pass
```

Разработанный класс Матриц должен проходить следующие тест-кейсы:

```
def test_matrix_operations():
    # Тестирование создания и вывода матрицы
    m1 = Matrix(2, 2)
    assert m1.rows == 2 and m1.cols == 2, "Ошибка: размеры матрицы
некорректны"

    # Тестирование сложения матриц
    m2 = Matrix(2, 2)
    m1.data = [[1, 2], [3, 4]]
    m2.data = [[5, 6], [7, 8]]
    result = m1 + m2
    assert result.data == [[6, 8], [10, 12]], "Ошибка: сложение матриц
выполнено неправильно"
```

```
    # Тестирование вычитания матриц
    result = m1 - m2
    assert result.data == [[-4, -4], [-4, -4]], "Ошибка: вычитание матриц выполнено неправильно"

    # Тестирование умножения матриц
    m3 = Matrix(2, 2)
    m3.data = [[1, 0], [0, 1]]
    result = m1 * m3
    assert result.data == [[1, 2], [3, 4]], "Ошибка: умножение матриц выполнено неправильно"

    # Тестирование транспонирования матрицы
    result = m1.transpose()
    assert result.data == [[1, 3], [2, 4]], "Ошибка: транспонирование матрицы выполнено неправильно"

    # Тестирование исключения при сложении матриц разного размера
    m4 = Matrix(3, 2)
    try:
        result = m1 + m4
    except MatrixDimensionError:
        print("Тест пройден: исключение при сложении матриц разного размера")
    else:
        assert False, "Ошибка: не выброшено исключение при сложении матриц разного размера"

    # Тестирование исключения при умножении матриц с несовместимыми размерами
    m5 = Matrix(3, 2)
    try:
        result = m1 * m5
    except MatrixDimensionError:
        print("Тест пройден: исключение при умножении матриц с несовместимыми размерами")
    else:
        assert False, "Ошибка: не выброшено исключение при умножении матриц с несовместимыми размерами"

    print("Все тесты успешно пройдены!")

test_matrix_operations()
```