* [Вычислительная геометрия на плоскости](https://foxford.ru/lessons/32286/conspects/1)
* [Создание структур Point и Vector для работы с точками и векторами (C++)](https://foxford.ru/lessons/32286/conspects/2)
* [Создание классов Point и Vector для работы с точками и векторами (Python)](https://foxford.ru/lessons/32286/conspects/3)

**Вычислительная геометрия на плоскости**

**Точки и вектора на плоскости**

Напомним несколько общеизвестных сведений из планиметрии.

Точки на плоскости можно задавать при помощи пары чисел (x,y). Расстояние от начала координат до точки P(x,y) легко находится по теореме Пифагора и равно x2+y2.

Вектора на плоскости также можно задавать при помощи двух координат. Длина вектора a→(x,y) вычисляется так же, как и расстояние от начала координат до точки: x2+y2.

Расстояние между двумя точками A1(x1,y1) и A2(x2,y2) вычисляется, как длина вектора A1A2→:  (x2−x1)2+(y2−y1)2.

Пусть дан вектор a→(x,y). Нормализованным вектором называется вектор единичной длины, сонаправленный данному. Его координаты есть (ax|a|,ay|a|).

**Радианная и градусная мера**

В математике принято измерять углы в радианах. Все тригонометрические функции в стандартной библиотеке языка Питон и в иных современных языках программирования используют только радианную меру углов.

Радианная мера угла — это такой способ измерения углов, при котором мерой угла считается длина дуги единичной окружности, которую вырезает центральный угол, равный данному. То есть угол в 360 градусов соответствует углу в 2π радиан. Если мера угла в градусах равна d, а мера угла в радианах равна r, то можно составить пропорцию: d360=r2π, откуда можно вывести формулы для перевода из градусов в радианы.

В компиляторе GNU C/C++ в заголовочном файле cmath есть константа

**M\_PI**

, равная значению числа π, а также ряд других констант.

В Visual C++ для того, чтобы использовать эти константы, нужно определить макрос

**\_USE\_MATH\_DEFINES**

 до подключения заголовочного файла

**cmath**

:

**#define \_USE\_MATH\_DEFINES**

**#include <cmath>**

 В библиотеке math языка Питон есть функции для перевода углов из градусов в радианы и обратно.

**Полярные координаты и полярный угол точки**

В некоторых задачах удобно использовать полярную систему координат. В полярной системе координат точка задается двумя числами — расстоянием до начала координат (обычно обозначается r) и углом между осью OX и радиус-вектором точки (обычно обозначается φ и называется полярным углом).

При этом величина r является неотрицательным, а величина φ обычно принимает значения от 0 до 2π (или от −π до π), но можно расширить возможные значения полярного угла на множество всех действительных чисел, в этом случае одной точке будет соответствовать бесконечно много полярных координат.

Для перевода координат из полярной системы в декартову можно использовать формулы для проекции вектора на оси координат:

x=rcos⁡φ

y=rsin⁡φ

Для вычисления значения r по декартовым координатам можно вычислить синус, косинус или тангенс полярного угла по координатам, а затем воспользоваться обратными тригонометрическими функциями (арксинус, арккосинус, арктангенс). Но так как у разных углов тригонометрические функции могут совпадать (например, sin⁡π/4=sin⁡3π/4), то одной триногометрической функции недостаточно, нужно рассмотреть случаи нахождения ответа в разных координатных четвертях.

Для удобства в языке программирования Питон и во всех современных языках программирования есть специальная функция atan2(y,x), которая возвращает полярный угол точки с координатами (x,y). Возвращаемым значением является полярный угол в радианах на интервале (−π,π). Обратите внимание, что первым параметром функции atan2 является значение y, а вторым — значение x.

**Скалярное произведение векторов**

Скалярное произведение двух векторов a¯(ax,ay) и b¯(bx,by) определяется как (a¯,b¯)=|a¯|⋅|b¯|⋅cos⁡φ, где φ -  угол между векторами.

Выражение скалярного произведения через координаты: (a¯,b¯)=axbx+ayby.

Свойства скалярного произведения:

1. Скалярное произведение коммутативно: (a¯,b¯)=(b¯,a¯).
2. Скалярное произведение положительно, если угол между векторами — острый, отрицательно — если тупой и равно 0, если вектора перпендикулярны.
3. Скалярное произведение линейно: (a+b¯,c¯)=(a¯,c¯)+(b¯,c¯), (ka¯,c¯)=k(a¯,c¯). Аналогичные утверждение верны и для второго аргумента скалярного произведения.

Скалярное произведение необходимо использовать, когда нужно проверить два вектора (два отрезка, две прямые) на перпендикулярность или исследовать угол между векторами (является ли он острым или тупым).

#### Создание структур Point и Vector для работы с точками и векторами (C++)

### Создание структуры Point, поля структуры

Для удобства работы с геометрическими примитивами (точками и векторами) полезно хранить точку (вектор) не в виде двух отдельных переменных (координат точки), а в виде одного объекта, содержащего две переменные-координаты. Это можно сделать в рамках концепции объектно-ориентированного программирования — необходимо создать класс «Точка», объекты которого будут содержать два поля — координату x и координату y. В этом случае можно работать с объектом «точка», как с одним целым, а можно и работать с отдельными координатами точки.

В языке C для этого использовалось понятие "структуры", в языке C++ помимо структур появилось понятие "класс". В современном стандарте языка C++ разница между структурами и классами невелика (практически она заключается только в правах доступа к полям и методам, у структур по умолчанию используется public-доступ, у классов - private-доступ), мы будем везде использовать структуры.

Для хранения точек и векторов необходима структура, содержащая два поля: x и y. Подобная структура определяется так:

**struct Point**

**{**

**double x, y;**

**};**

Обратите внимание, точка с запятой после закрывающей фигурной скобки обязательна!

После этого можно объявлять отдельные объекты структуры Point следующим образом:

**Point P, Q;**

Для доступа к отдельным полям класса используется так называемая «дот-нотация», когда название поля пишется через точку после идентификатора переменной. Например, задать для точки координаты можно так:

**P.x = 5;**

**P.y = -10;**

### Конструкторы

Если полям структуры не присвоить явно какие-либо значения, то в них будет храниться "мусор", как в случае локальных переменных. Поэтому желательно реализовать специальный метод-"конструктор", который будет автоматически вызываться при объявлении переменной типа "структура". Назначение конструктора - инициализация полей структуры.

Конструктор - это метод, название которого совпадает с названием самой структуры. Вот пример объявления структуры Point вместе с конструктором, который инициализирует поля структуры нулями:

**struct Point**

**{**

**double x, y;**

**Point()**

**{**

**x = 0;**

**y = 0;**

**}**

**};**

Обратите внимание, конструктор не возвращает никакого значения, даже слово void нельзя писать в определении конструктора.

Теперь при объявлении

**Point P, Q;**

у объявленных переменных поля x и y будут равны 0.

Можно объявлять несколько конструкторов, отличающихся разным набором параметров. Например, добавим конструктор, который принимает на вход два числа — координаты точки (этот конструктор также нужно написать внутри описания структуры):

**Point(double a, double b)**

**{**

**x = a;**

**y = b;**

**}**

Теперь можно создавать объекты структуры Point, задавая координаты точки. Например, так:

**Point P(1, -1);**

Другой полезный пример конструктора — объявим структуру "вектор". Хотя структуры "вектор" и "точка" обе имеют два поля x и y, может показаться, что для этих понятий достаточно использовать одну структуру (т.к. каждую точку можно рассматривать, как радиус-вектор). Но лучше для этих понятий сделать две разные структуры, например, у вектора может быть конструктор от двух точек:

**struct Vector**

**{**

**double x, y;**

**Vector(Point A, Point B)**

**{**

**x = B.x - A.x;**

**y = B.y - A.y;**

**}**

**};**

Теперь можно объявлять векторы следующим образом:

**Point P, Q;**

**…**

**Vector PQ(P, Q);**

и сразу становится понятно, что вектор PQ — это вектор, проведенный из точки P в точку Q.

### Методы

У структур также бывают "методы" — это функции, вызывающиеся непосредственно для экземпляров структуры. Например, у точки может быть метод dist, который возвращает расстояние от начала координат до точки, аналогичный метод должен быть и у вектора. Методы можно создавать также внутри описания структуры, например, после конструкторов:

**double dist()**

**{**

**return sqrt(x \* x + y \* y);**

**}**

Вызывается метод так:

**cout << P.dist() << endl;**

### Переопределение арифметических операций

Помимо методов можно также переопределять операторы. Например, хочется чтобы можно было складывать вектора при помощи операции "+" вот таким образом:

**Vector AB, BC, AC;**

**…**

**AC = AB + BC;**

Для этого необходимо определить функцию со специальным названием "operator+", которая принимает на вход два аргумента типа Vector и возвращает значение типа Vector. Это необходимо сделать путем объявления отдельной функции, то есть вне описания структуры (есть и другой способ, объявление метода структуры, но мы не будем его рассматривать). Функция получает два аргумента типа Vector (левый и правый операнд) и возвращает значение типа Vector;

Вот пример объявления такой функции:

**Vector operator+ (Vector a, Vector b)**

**{**

**return Vector(a.x + b.x, a.y + b.y);**

**}**

Аналогично определим, например, операцию умножения вектора на число:

**Vector operator\* (Vector a, double d)**

**{**

**return Vector(a.x \* d, a.y \* d);**

**}**

**Vector operator\* (double d, Vector a)**

**{**

**return Vector(a.x \* d, a.y \* d);**

**}**

Заметим, что для того, чтобы можно было умножать вектор на число и число на вектор, то есть менять порядок операндов, необходимо определить две функции.

Аналогично можно определить любые другие операции для точек и векторов. Но можно только переопределять существующие в синтаксисе языка операции, нельзя определить новые операции, которых нет в синтаксисе языка (то есть нельзя, например, определить операцию, обозначаемую знаком "@").

### Переопределение операций ввода-вывода

Переопределение операций ">>" и "<<" используется для того, чтобы можно было легко считывать и выводить точки и вектора. Определим, например, операции ввода-вывода для точек:

**istream & operator>> (istream & in, Point & P)**

**{**

**in >> P.x >> P.y;**

**return in;**

**}**

**ostream & operator<< (ostream & out, Point & P)**

**{**

**out << P.x << " " << P.y;**

**return out;**

**}**

Теперь можно считывать координаты точки P из потока ввода (например, из cin, но можно и из файла) при помощи:

**cin >> P;**

а выводить координаты точки (через пробел) при помощи:

**cout << P;**

#### Создание классов Point и Vector для работы с точками и векторами (Python)

Для удобства работы с геометрическими примитивами (точками и векторами) полезно хранить точку (вектор) не в виде двух отдельных переменных (координат точки), а в виде одного объекта, содержащего две переменные-координаты. Это можно сделать в рамках концепции объектно-ориентированного программирования — необходимо создать класс «Точка», объекты которого будут содержать два поля — координату x и координату y. В этом случае можно работать с объектом «точка», как с одним целым, а можно и работать с отдельными координатами точки.

Самый простой способ объявить класс в языке Питон:

**class Point:**

**pass**

После этого можно объявлять отдельные объекты класса Point следующим образом:

**P = Point()**

**Q = Point()**

Для доступа к отдельным полям класса используется так называемая «дот-нотация», когда название поля пишется через точку после идентификатора переменной. Например, задать для точки координаты можно так:

**P.x = 5**

**P.y = -10**

Обратите внимание, что запись вида

**Q = P**

не создает новый объект, а просто делает Q ссылкой на тот же объект, что и P. Поэтому если изменить у объекта P значение какого-нибудь поля, то и у объекта Q это значение изменится. Для создания копии объекта P, скопировав значения всех его полей, нужно использовать функцию copy из модуля copy:

**import copy**

**Q = copy.copy(P)**

Можно сделать так, чтобы у объекта автоматически создавались какие-нибудь поля, например, чтобы при создании точки у нее создавались поля x и y. Для этого необходимо определить конструктор объекта — метод, автоматически вызывающийся при создании объекта. Конструктором является метод, имеющий специальное название \_\_init\_\_

Конструкторы и прочие методы определяются внутри класса. Например:

**class Point:**

**def \_\_init\_\_(self):**

**self.x = 0**

**self.y = 0**

У конструктора и любого другого метода класса первым передаваемым параметром обязательно должен быть параметр self, который является ссылкой на объект, для которого будет вызван этот метод. При этом при вызове метода не нужно этот параметр передавать. Вот пример объявления метода dist, возвращающего расстояние от точки до начала координат:

**def dist(self):**

**return (self.x \*\* 2 + self.y \*\* 2) \*\* 0.5**

Вызывается метод так:

**print(P.dist())**

Конструктор может принимать и какие-нибудь параметры. Например, рассмотрим следующий конструктор:

**class Point:**

**def \_\_init\_\_(self, x=0, y=0):**

**self.x = 0**

**self.y = 0**

Этому конструктору можно передать в качестве параметра два числа — координаты точки. Например, можно его вызывать так:

**P = Point(2, 6)**

Если же не передавать эти два параметра, а вызвать конструктор без параметров, то значения полей x и y будут проинициализированы по умолчанию числом 0.

Есть и другие стандартные методы, которые можно определить в описании класса. Например, метод

**\_\_str\_\_**

возвращает строку, являющуюся описанием объекта в том виде, в котором его удобно будет воспринимать человеку. Здесь не нужно выводить имя конструктора, можно, например, просто вернуть строку с содержимым всех полей:

**class Point:**

**def \_\_str\_\_(self):**

**return str(self.x) + ' ' + str(self.y)**

Метод str будет вызываться, когда вызывается функция str от данного объекта, например, str(P). То есть создавая метод str вы даете указание Питону, как преобразовывать данный объект к типу str.

Поскольку функция print использует именно функцию str для вывода объекта на экран, то определение метода str позволит выводить объекты на экран удобным способом: при помощи print, например, можно будет писать:

**print(P)**

Если рассматривать точку как радиус-вектор, то хотелось бы определить для точек операцию +, чтобы точки можно было складывать столь же удобно, как и числа или строки. Например, чтобы можно было записать так:

**A = Point(1, 2)**

**B = Point(3, 4)**

**C = A + B**

Для этого необходимо перегрузить операцию +: определить функцию, которая будет использоваться, если операция + будет вызвана для объекта класса Point. Для этого нужно определить метод add класса Point, у которого два параметра: неявная ссылка self на экземпляр класса, для которого она будет вызвана (это левый операнд операции +) и явная ссылка other на правый операнд:

**def \_\_add\_\_(self, other):**

**return Point(self.x + other.x, self.y + other.y)**

Теперь при вызове оператора A + B Питон вызовет метод A.add(B), то есть вызовет указанный метод, где self = A, other = B.

Аналогично можно определить и оставшиеся операции. Вот (неполный) список возможных операций, которые можно переопределить для объектов:

|  |  |
| --- | --- |
| **Метод** | **Пример вызова** |
| **Операторы сравнения** |  |
| \_\_lt\_\_(self, other) | x < y |
| \_\_le\_\_(self, other) | x <= y |
| \_\_eq\_\_(self, other) | x == y |
| \_\_ne\_\_(self, other) | x != y |
| \_\_gt\_\_(self, other) | x > y |
| \_\_ge\_\_(self, other) | x >= y |
| **Арифметические операторы** |  |
| **Сложение** |  |
| \_\_add\_\_(self, other) | x + y |
| \_\_radd\_\_(self, other) | y + x |
| \_\_iadd\_\_(self, other) | x += y |
| **Вычитание** |  |
| \_\_sub\_\_(self, other) | x - y |
| \_\_rsub\_\_(self, other) | y - x |
| \_\_isub\_\_(self, other) | x -= y |
| **Умножение** |  |
| \_\_mul\_\_(self, other) | x \* y |
| \_\_rmul\_\_(self, other) | y \* x |
| \_\_imul\_\_(self, other) | x \*= y |
| **Деление** |  |
| \_\_truediv\_\_(self, other) | x / y |
| \_\_rtruediv\_\_(self, other) | y / x |
| \_\_itruediv\_\_(self, other) | x /= y |
| **Целочисленное деление** |  |
| \_\_floordiv\_\_(self, other) | x // y |
| \_\_rfloordiv\_\_(self, other) | y // x |
| \_\_ifloordiv\_\_(self, other) | x //= y |
| \_\_divmod\_\_(self, other) | divmod(x,y) |
| **Остаток** |  |
| \_\_mod\_\_(self, other) | x % y |
| \_\_rmod\_\_(self, other) | y % x |
| \_\_imod\_\_(self, other) | x %=y |
| **Возведение в степень** |  |
| \_\_pow\_\_(self, other) | x \*\* y |
| \_\_rpow\_\_(self, other) | y \*\* x |
| \_\_ipow\_\_(self, other) | x \*\*= y |
| **Отрицание, модуль** |  |
| \_\_pos\_\_(self) | +x |
| \_\_neg\_\_(self) | -x |
| \_\_abs\_\_(self) | abs(x) |
| **Преобразование к стандартным типам** |  |
| \_\_int\_\_(self) | int(x) |
| \_\_float\_\_(self) | float(x) |
| \_\_str\_\_(self) | str(x) |
| \_\_round\_\_(self, digits = 0) | round(x, digits) |