**Ճարտարապետության և շինարարության Հայաստանի ազգային համալսարան**

ՌԵՖԵՐԱՏ

Ֆակուլտետ՝ Կառավարման և տեխնոլոգիայի

Մասնագիտություն՝ Ինֆորմատիկա (համակարգչային գիտություն)

Կուրս՝ II, բակալավրիատ, հեռակա

Թեմա՝ «Օբյեկտ-կողմնորոշված ծրագրավորման տարրերի և կառույցների հիմանական տարբերությունները Python և C++ ծրագրավորման լեզուների միջև»

Ուսանող՝ Տիգրան Վահրամի Գրիգորյան

Դասախոս՝ Մանուսաջյան Անահիտ

Երևան 2025թ.

**Օբյեկտ-կողմնորոշված ծրագրավորման տարրերի և կառույցների հիմանական տարբերությունները Python և C++ ծրագրավորման լեզուների միջև**

Որպես ուսումնասիրության հիմք ընդունենք հետևյալ խնդիրը․ անհրաժեշտ է գրել ծրագիր, որը պետք է ներկայացնի Point դասը դեկարտյան հարթության վրա՝ ապահովելով համապատասխան մեթոդներ և ֆունկցիաներ հարթության վրա տարբեր գործողություններ կատարելու համար։

Point դասի ատրիբուտներ

* x՝ հարթության վրա x կոորդինատը:
* y՝ հարթության վրա y կոորդինատը:
* Առանց պարամետրի կոնստրուկտոր՝ ստեղծում է կետ հարթության սկզբնակետում։
* Երկու պարամետրով կոնստրուկտոր՝ կոորդինատները ընդունում է որպես պարամետրեր։
* Պատճենի կոնստրուկտոր։
* Դեստրուկտոր։
* x և y ատրիբուտների համար առանձին getter-ներ և setter-ներ։
* x և y ատրիբուտների համար միասնական getter և setter։
* Հարթության քառորդին կետի պատկանելության վերադարձման մեթոդ։
* Կետերի համեմատման մեթոդներ։
* Երկու կետերի միջև հեռավորության հաշվման մեթոդ։
* Սկզբնակետից կետի հեռավորության հաշվման մեթոդ։
* Ox և Oy առանցքների նկատմամվ կետի զուգահեռ տեղաշարժման մեթոդներ։

Եկեք մեկ առ մեկ ուսումնասիրենք անհրաժեշտ կառույցները և զուգահեռ համեմատենք դրանց իրականացումները Python-ում և C++-ում։

**Կիրառված գրադարաններ և մոդուլներ՝**

**Python`**

from \_\_future\_\_ import annotations *# հատուկ կարգադրություն Python-ի compiler-ի համար, կոդի անոտացիայի հետ կապված ֆունկցիոնալությունը ակտիվացնելու համար*

from math import sqrt *# ֆունկցիա թվից քառակուսային արմատ հաշվելու համար*

from typing import ClassVar, Literal *# հավելյալ կառույցներ կոդի անոտացիայի համար նախատեսված*

**C++`**

**#include <iostream> *// հիմնական ներմուծման/արտածման ֆունկցիաները (cout, cin և այլն)՝ տվյալների մուտքագրում և ելք արտածելու համար։***

**#include <cmath> *// մաթեմատիկական ֆունկցիաներ՝ օրինակ՝ sqrt(), pow(), sin(), cos() և այլն։***

**#include <vector> *// վեկտորներ (dynamic arrays)՝ տվյալների մեջ փոփոխություններ կատարելու և տարրեր ավելացնելու/հեռացնելու հնարավորությամբ։***

**#include <string> *// string-երի հետ աշխատանք***

Կոնստրուկտորներ և հիմնական ատրիբուտների վերագրում

**Python`**

class Point:

points: ClassVar[list[Point]] = []

def \_\_init\_\_(self, x\_coord: float = 0, y\_coord: float = 0) -> None:

self.\_x\_coord = x\_coord

self.\_y\_coord = y\_coord

print(f"Point created, coordinates: {self.coordinates}")

Point.points.append(self)

....

**C++`**

**class Point**

**{**

**private:**

**float x\_coord;**

**float y\_coord;**

**public:**

***// A class-level list to hold all points***

**static std::vector<Point\*> points;**

***// Constructor***

**Point(float x = 0, float y = 0) : x\_coord(x), y\_coord(y) {**

**points.push\_back(this);**

**std::cout << "Point created, coordinates: (" << x\_coord << ", " << y\_coord << ")\n";**

**}**

....

**}**

***// Definition of static class member***

**std::vector<Point\*> Point::points;**

Ինչպես տեսնում ենք, երկու իրականացումներում էլ class-ները ունեն հատուկ "points" ատրիբուտ/member, որը C++-ում vector է, իսկ Python-ում՝ list, որում պահվում են բոլոր կետերը՝ սկսած դրանց ստեղծման պահից։  
  
 Python-ում կոնստրուկտորի դերը տանում է \_\_init\_\_ dunder մեթոդը, և ի տարբերություն C++-ի, դասական ձևերով հնարավոր չէ ունենալ ավելի քան մեկ \_\_init\_\_ մեթոդ։

Բացի այդ, C++-ի իրականացման մեջ տեսնում ենք, որ առանձին հայտարարվում են կոորդինատների համար նախատեսված երկու float տիպի private member՝ x\_coord և y\_coord, սակայն Python-ում հնարավոր չէ ստեղծել ամբողջովին private ատրիբուտներ։ Այդ իսկ պատճառով համարվում է, որ մեկ underscore-ով ( \_ ) սկսվող ատրիբուտները private են և չպետք է կիրառվեն օբյեկտի կամ դասի սահմաններից դուրս։ Հետևաբար, տեսնում ենք, որ երկու ատրիբուտի արժեքները վերագրվում են կրկին float տիպի \_x\_coord և \_y\_coord։

Մեկ այլ նշանակալի տարբերություն այն է, որ օբյեկտին կցված կառույցներում C++-ում օբյեկտին հասանելիությունը տրվում է this փոփոխականի միջոցով, իսկ Python-ում օգտագործվում է self գրառմամբ փոփոխականը:

Բացի այդ, տեսնում ենք, որ երկու դեպքում էլ x և y արգումենտների համար սահմանված են կոորդինատական համակարգի սկզբնակետին համապատասխանող լռելյայն արժեքներ, որի շնորհիվ հնարավոր է ստեղծել Point դասի օբյեկտ կամ առանց կոորդինատ փոխանցելու, կամ միայն մեկ կոորդինատ փոխանցելով, կամ էլ երկու կոորդինատները միաժամանակ։

Պատճենի կոնստրուկտոր

**Python`**

class Point:

....

def \_\_copy\_\_(self) -> Point:

return Point(\*self.coordinates)

....

**C++`**

class Point

{

....

public:

....

*// Copy constructor*

Point(const Point& oth)

{

x\_coord = oth.x\_coord;

y\_coord = oth.y\_coord;

points.push\_back(this);

};

....

};

Ինչպես տեսնում ենք, Python-ում copy կոնստրուկտորի դերը տանում է \_\_copy\_\_ dunder մեթոդը, որտեղ օգտագործելով ներկայիս օբյեկտի կոորդինատները՝ ստեղծվում և վերադառնում է նոր օբյեկտ, կրկին անցնելով օբյեկտի initialisation-ի փուլով։ Ի տարբերություն դրան, C++-ում օբյեկտը պատճենելու համար անհրաժեշտ չէ դիմել հիմնական կոնստրուկտորին, քանի որ անհրաժեշտ ատրիբուտները անմիջապես վերագրվում են պատճենված օբյեկտին։ Հարկ է նշել, որ oth փոփոխականը reference է պատճենված օբյեկտին։

Դեստրուկտոր

**Python`**

class Point:

    ....

def \_\_del\_\_(self) -> None:

Point.points.remove(self)

print(f"Point with '{self.coordinates}' deleted.")

    ....

**C++`**

class Point

{

....

public:

    ....

*// Destructor*

~Point() {

points.erase(std::remove(points.begin(), points.end(), this), points.end());

std::cout << "Point with coordinates (" << x\_coord << ", " << y\_coord << ") deleted.\n";

}

....

};

Այս դեպքում տեսնում ենք, որ Python-ում դեստրուկտորի դերը տանում է \_\_del\_\_ dunder մեթոդը։ Սակայն հարկ է նշել, որ այդ մեթոդը շատ հազվադեպ է օգտագործվում production code-ում, քանի որ Python-ի garbage collection մեխանիզմը չի երաշխավորում, որ այն կկանչվի հենց այն պահին, երբ օգտատերը ջնջում է օբյեկտը։ Հետևաբար, դա կարող է հաճախ հանգեցնել undefined behaviour-ի։

x և y ատրիբուտների համար getter-ներ և setter-ներ

**Python`**

class Point:

@property

def x\_coord(self) -> float:

return self.\_x\_coord

@x\_coord.setter

def x\_coord(self, value: float) -> None:

self.\_x\_coord = value

@property

def y\_coord(self) -> float:

return self.\_y\_coord

@y\_coord.setter

def y\_coord(self, value: float) -> None:

self.\_y\_coord = value

@property

def coordinates(self) -> tuple[float, float]:

return self.x\_coord, self.y\_coord

@coordinates.setter

def coordinates(self, coord: tuple[float, float]) -> None:

self.x\_coord, self.y\_coord = coord

  ....

**C++`**

class Point

{

....

public:

    ....

*// Getters and setters for x\_coord*

float get\_x() const { return x\_coord; }

void set\_x(const float value) { x\_coord = value; }

*// Getters and setters for y\_coord*

float get\_y() const { return y\_coord; }

void set\_y(const float value) { y\_coord = value; }

*// Get coordinates as a pair*

std::pair<float, float> get\_coordinates() const {

return std::make\_pair(x\_coord, y\_coord);

}

*// Set coordinates*

void set\_coordinates(const float x, const float y) {

set\_x(x);

set\_y(y);

}

....

};

Ինչպես տեսնում ենք, Python-ը getter և setter մեթոդների համար տրամադրում է հատուկ property և \*.setter դեկորատորներ, որոնց շնորհիվ մեթոդների տակ թաքնված ատրիբուտները հասանելի են լինում սովորական ատրիբուտների համար նախատեսված դիմելաձևերով։ Բացի այդ, հարկ է նշել, որ C++-ում երկու կոորդինատները միասնականորեն վերադարձնելու համար կիրառվում է pair կառույցը, որը Python-ում գոյություն չունի, և այդ պատճառով փոխարինվում է tuple-ով։

Հարթության քառորդին կետի պատկանելության վերադարձում

**Python`**

class Point:

@property

def sector(self) -> Literal[

"Point is on both Axis",

"Point is on X Axis",

"Point is on Y Axis",

"Point is in First Sector",

"Point is in Second Sector",

"Point is in Third Sector",

"Point is in Fourth Sector",

"ERROR",

]:

match self.coordinates:

case x\_coord, y\_coord if x\_coord == 0 and y\_coord == 0:

return "Point is on both Axis"

case x\_coord, \_ if x\_coord == 0:

return "Point is on X Axis"

case \_, y\_coord if y\_coord == 0:

return "Point is on Y Axis"

case x\_coord, y\_coord if x\_coord > 0 and y\_coord > 0:

return "Point is in First Sector"

case x\_coord, y\_coord if x\_coord < 0 and y\_coord > 0:

return "Point is in Second Sector"

case x\_coord, y\_coord if x\_coord < 0 and y\_coord < 0:

return "Point is in Third Sector"

case x\_coord, y\_coord if x\_coord > 0 and y\_coord < 0:

return "Point is in Fourth Sector"

case \_:

return "ERROR"

    ....

**C++`**

class Point

{

....

public:

    ....

*// Sector calculation*

std::string sector() const {

if (x\_coord == 0 && y\_coord == 0)

return "Point is on both Axis";

else if (x\_coord == 0)

return "Point is on X Axis";

else if (y\_coord == 0)

return "Point is on Y Axis";

else if (x\_coord > 0 && y\_coord > 0)

return "Point is in First Sector";

else if (x\_coord < 0 && y\_coord > 0)

return "Point is in Second Sector";

else if (x\_coord < 0 && y\_coord < 0)

return "Point is in Third Sector";

else if (x\_coord > 0 && y\_coord < 0)

return "Point is in Fourth Sector";

else

return "ERROR";

}

    ....

};

Ինչպես տեսնում ենք, կոդի մակարդակով այս դեպքում միակ տարբերությունը այն է, որ Python-ում օգտագործվել է switch/match case, իսկ C++-ում՝ ոչ։ Այնուամենայնիվ, պետք է նշել, որ սա որպես այդպիսի ոչինչ չի փոխում, քանի որ C++-ի դեպքում այս կտորը առանց այդ էլ կոմպիլյատորի կողմից օպտիմիզացվելու է վերածվելով jump-table-ի, իսկ Python-ի դեպքում այս տիպի միկրոօպտիմիզացիաները անտեղի են և ունեն շատ փոքր ազդեցություն։

Կետերի համեմատման մեթոդներ

**Python`**

**class Point:**

**....**

**def \_\_eq\_\_(self, other: Point) -> bool:**

**return self.coordinates == other.coordinates**

**....**

**C++`**

class Point

{

....

public:

    ....

*// Equality and Non-Equality operators overloading*

bool operator==(const Point& other) const {

return x\_coord == other.get\_x() && y\_coord == other.get\_y();

}

bool operator != (const Point& other) const

{

return ! (\*this == other);

}

    ....

};

Եվ կրկին տեսնում ենք, որ Python-ում մեծ քանակությամբ օպերացիաներ տեղի են ունենում dunder method-ների շնորհիվ։ Կոնկրետ այս դեպքում \_\_eq\_\_ մեթոդը կատարում է հավասարության պայմանին բավարարելու հաշվարկը։ Ի տարբերություն դրան, C++-ում հարկավոր է overload անել հավասարության և անհավասարության օպերատորները նույն արդյունքը ստանալու համար։ Բացի այդ, հարկ է նշել, որ Python-ում միայն հավասարության մեթոդը սահմանելով անմիջապես սկսում է գործել նաև անհավասարության \_\_neq\_\_ մեթոդը, սակայն հաճախ այն նույնպես առանձին գրվում է, քանի որ Python MRO-ը (Method Resolution Order) կարող է շատ երկար և բարդ լինել, ինչի հետևանքով օպերացիան անիմաստ կդանդաղի։

Երկու կետերի միջև հեռավորության հաշվման մեթոդ

**Python`**

class Point:

    ....

@staticmethod

def distance\_between\_points(pt1: Point, pt2: Point) -> float:

x\_dist = abs(pt1.x\_coord - pt2.x\_coord)

y\_dist = abs(pt1.y\_coord - pt2.y\_coord)

return sqrt(x\_dist\*\*2 + y\_dist\*\*2)

    ....

**C++`**

class Point

{

....

public:

    ....

*// Distance between two points*

static float distance\_between\_points(const Point& pt1, const Point& pt2) {

float x\_dist = std::abs(pt1.get\_x() - pt2.get\_x());

float y\_dist = std::abs(pt1.get\_y() - pt2.get\_y());

return std::sqrt(x\_dist \* x\_dist + y\_dist \* y\_dist);

}

    ....

};

Ինչպես տեսնում ենք, միակ տարբերությունը այս երկու իրականացումներում ստատիկ մեթոդի հայտարարման ձևն է։ Ի տարբերություն C++-ի, Python-ում առանձին “static” keyword գոյություն չունի, և նրա դերը տանում է մեթոդների համար նախատեսված staticmethod դեկորատորը։

Սկզբնակետից կետի հեռավորության հաշվման մեթոդ

**Python`**

class Point:

    ....

@property

def distnance\_from\_0(self) -> float:

return sqrt(self.x\_coord\*\*2 + self.y\_coord\*\*2)

    ....

**C++`**

class Point

{

....

public:

    ....

*// Calculate distance from origin (0, 0)*

float distance\_from\_0() const {

return std::sqrt(x\_coord \* x\_coord + y\_coord \* y\_coord);

}

    ....

};

Այս դեպքի միակ տարբերությունը Python-ում property-ի կիրառումն է, իսկ մնացած բոլոր ասպեկտներում տարբերություն չկա։

Ox և Oy առանցքների նկատմամբ կետի զուգահեռ

տեղաշարժման մեթոդներ

**Python`**

**class Point:**

**....**

**def move\_on\_x(self, value: float) -> None:**

**self.\_x\_coord += value**

**def move\_on\_y(self, value: float) -> None:**

**self.\_y\_coord += value**

**....**

**C++`**

class Point

{

....

public:

    ....

void move\_on\_x(const float value) { x\_coord += value; }

void move\_on\_y(const float value) { y\_coord += value; }

    ....

};

Կարելի է ասել, որ ոչ մի նշանակալի տարբերություն չկա։

Արդյունք

**Python`**

from \_\_future\_\_ import annotations

from math import sqrt

from typing import ClassVar, Literal

class Point:

points: ClassVar[list[Point]] = []

def \_\_init\_\_(self, x\_coord: float = 0, y\_coord: float = 0) -> None:

self.\_x\_coord = x\_coord

self.\_y\_coord = y\_coord

print(f"Point created, coordinates: {self.coordinates}")

Point.points.append(self)

def \_\_del\_\_(self) -> None:

Point.points.remove(self)

print(f"Point with '{self.coordinates}' deleted.")

def \_\_eq\_\_(self, other: Point) -> bool:

return self.coordinates == other.coordinates

def \_\_copy\_\_(self) -> Point:

return Point(\*self.coordinates)

@property

def x\_coord(self) -> float:

return self.\_x\_coord

@x\_coord.setter

def x\_coord(self, value: float) -> None:

self.\_x\_coord = value

def move\_on\_x(self, value: float) -> None:

self.\_x\_coord += value

@property

def y\_coord(self) -> float:

return self.\_y\_coord

@y\_coord.setter

def y\_coord(self, value: float) -> None:

self.\_y\_coord = value

def move\_on\_y(self, value: float) -> None:

self.\_y\_coord += value

@property

def coordinates(self) -> tuple[float, float]:

return self.x\_coord, self.y\_coord

@coordinates.setter

def coordinates(self, coord: tuple[float, float]) -> None:

self.x\_coord, self.y\_coord = coord

@staticmethod

def distance\_between\_points(pt1: Point, pt2: Point) -> float:

x\_dist = abs(pt1.x\_coord - pt2.x\_coord)

y\_dist = abs(pt1.y\_coord - pt2.y\_coord)

return sqrt(x\_dist\*\*2 + y\_dist\*\*2)

@property

def sector(

self,

) -> Literal[

"Point is on both Axis",

"Point is on X Axis",

"Point is on Y Axis",

"Point is in First Sector",

"Point is in Second Sector",

"Point is in Third Sector",

"Point is in Fourth Sector",

"ERROR",

]:

match self.coordinates:

case x\_coord, y\_coord if x\_coord == 0 and y\_coord == 0:

return "Point is on both Axis"

case x\_coord, \_ if x\_coord == 0:

return "Point is on X Axis"

case \_, y\_coord if y\_coord == 0:

return "Point is on Y Axis"

case x\_coord, y\_coord if x\_coord > 0 and y\_coord > 0:

return "Point is in First Sector"

case x\_coord, y\_coord if x\_coord < 0 and y\_coord > 0:

return "Point is in Second Sector"

case x\_coord, y\_coord if x\_coord < 0 and y\_coord < 0:

return "Point is in Third Sector"

case x\_coord, y\_coord if x\_coord > 0 and y\_coord < 0:

return "Point is in Fourth Sector"

case \_:

return "ERROR"

@property

def distnance\_from\_0(

self,

) -> float:

return sqrt(self.x\_coord\*\*2 + self.y\_coord\*\*2)

**C++`**

#include <iostream>

#include <cmath>

#include <vector>

#include <string>

class Point

{

private:

float x\_coord;

float y\_coord;

public:

*// A class-level list to hold all points*

static std::vector<Point\*> points;

*// Constructor*

Point(float x = 0, float y = 0) : x\_coord(x), y\_coord(y) {

points.push\_back(this);

std::cout << "Point created, coordinates: (" << x\_coord << ", " << y\_coord << ")\n";

}

*// Destructor*

~Point() {

points.erase(std::remove(points.begin(), points.end(), this), points.end());

std::cout << "Point with coordinates (" << x\_coord << ", " << y\_coord << ") deleted.\n";

}

*// Copy constructor*

Point(const Point& oth)

{

x\_coord = oth.x\_coord;

y\_coord = oth.y\_coord;

points.push\_back(this);

};

*// Getters and setters for x\_coord*

float get\_x() const { return x\_coord; }

void set\_x(const float value) { x\_coord = value; }

void move\_on\_x(const float value) { x\_coord += value; }

*// Getters and setters for y\_coord*

float get\_y() const { return y\_coord; }

void set\_y(const float value) { y\_coord = value; }

void move\_on\_y(const float value) { y\_coord += value; }

*// Get coordinates as a pair*

std::pair<float, float> get\_coordinates() const {

return std::make\_pair(x\_coord, y\_coord);

}

*// Set coordinates*

void set\_coordinates(const float x, const float y) {

set\_x(x);

set\_y(y);

}

*// Distance between two points*

static float distance\_between\_points(const Point& pt1, const Point& pt2) {

float x\_dist = std::abs(pt1.get\_x() - pt2.get\_x());

float y\_dist = std::abs(pt1.get\_y() - pt2.get\_y());

return std::sqrt(x\_dist \* x\_dist + y\_dist \* y\_dist);

}

*// Calculate distance from origin (0, 0)*

float distance\_from\_0() const {

return std::sqrt(x\_coord \* x\_coord + y\_coord \* y\_coord);

}

*// Sector calculation*

std::string sector() const {

*// TODO(Tigran Grigoryan): maybe convert to switch case?*

if (x\_coord == 0 && y\_coord == 0)

return "Point is on both Axis";

else if (x\_coord == 0)

return "Point is on X Axis";

else if (y\_coord == 0)

return "Point is on Y Axis";

else if (x\_coord > 0 && y\_coord > 0)

return "Point is in First Sector";

else if (x\_coord < 0 && y\_coord > 0)

return "Point is in Second Sector";

else if (x\_coord < 0 && y\_coord < 0)

return "Point is in Third Sector";

else if (x\_coord > 0 && y\_coord < 0)

return "Point is in Fourth Sector";

else

return "ERROR";

}

*// Equality and Non-Equality operators overloading*

bool operator==(const Point& other) const {

return x\_coord == other.get\_x() && y\_coord == other.get\_y();

}

bool operator != (const Point& other) const

{

return ! (\*this == other);

}

};

*// Definition of static class member*

std::vector<Point\*> Point::points;