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 հիմնական տարբերությունները Python և C++ ծրագրավորման լեզուների միջև»

Ուսանող՝ Տիգրան Վահրամի Գրիգորյան

Դասախոս՝ Մանուսաջյան Անահիտ

Երևան 2025թ.

Օբյեկտ-կողմնորոշված ծրագրավորման տարրերի և կառույցների հիմնական 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 նկատմամբ 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՝

```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++՝

```cpp
#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`

```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++`

```cpp
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`

```python
class Point:
    ....
    def __eq__(self, other: Point) -> bool:
        return self.coordinates == other.coordinates
    ....
```

C++`

```cpp
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`

```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++`

```cpp
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`

```python
class Point:
    ....
    @property
    def distnance_from_0(self) -> float:
        return sqrt(self.x_coord**2 + self.y_coord**2)
    ....
```

C++`

```cpp
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`

```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++`

```cpp
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`

```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
```

```python
@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;
```