Data Structures

Lab01 - Abstract Data Types

---

1 Implement the following abstract data types

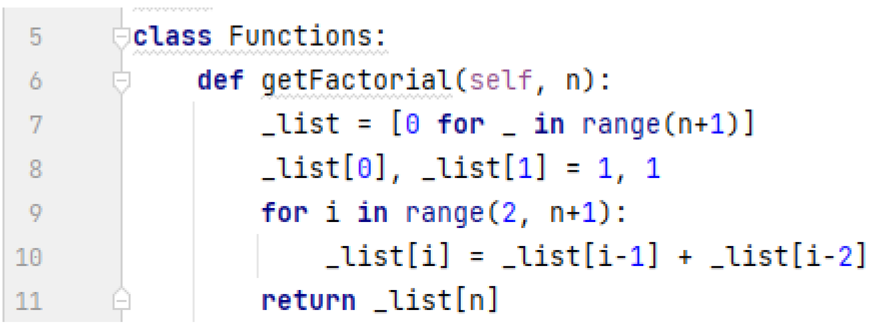
1. Functions Class

   The Functions class/Type that implements the following functions

   (a) Write a function (getFactorial). The factorial of a non-negative integer n is written n! (pronounced "n factorial") and is defined as follows:
   $n! = n \times (n-1) \times (n-2) \times \dots \times 1$

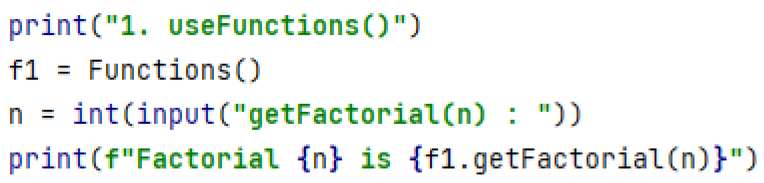
[code]

- **Functions.py**



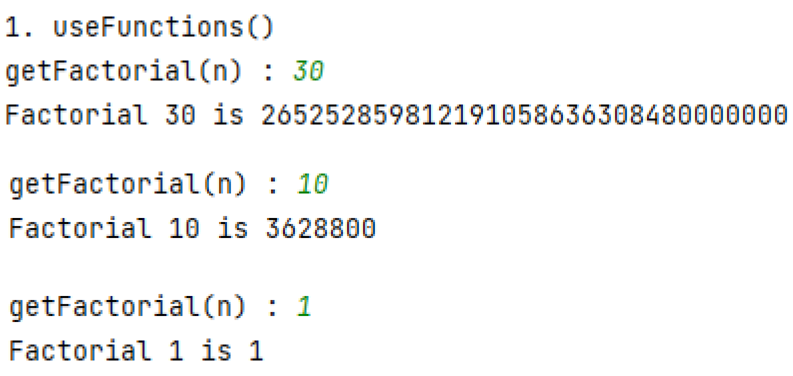
```python
class Functions:
    def getFactorial(self, n):
        _list = [0 for _ in range(n+1)]
        _list[0], _list[1] = 1, 1
        for i in range(2, n+1):
            _list[i] = _list[i-1] + _list[i-2]
        return _list[n]
```

- **Lab01Test.py**



```python
print("1. useFunctions()")
f1 = Functions()
n = int(input("getFactorial(n) : "))
print(f"Factorial {n} is {f1.getFactorial(n)}")
```

[result]



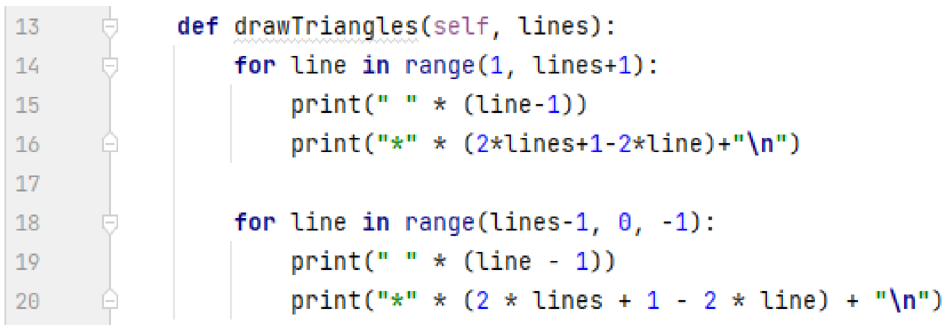
```
1. useFunctions()
getFactorial(n) : 30
Factorial 30 is 265252859812191058636308480000000

getFactorial(n) : 10
Factorial 10 is 3628800

getFactorial(n) : 1
Factorial 1 is 1
```

(b) Write a function drawTriangles() that prints the downward and upward triangles

```
*************
 ***********
  *********
   *******
    *****
     ***
      *

      *
     ***
    *****
   *******
  *********
 ***********
*************
```

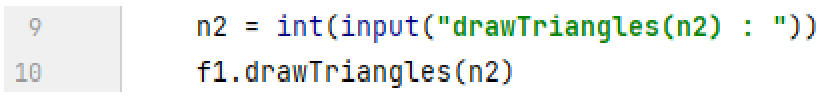
[code]

- **Functions.py**



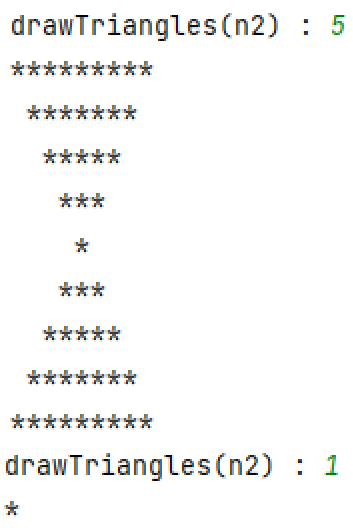
```python
def drawTriangles(self, lines):
    for line in range(1, lines+1):
        print(" " * (line-1))
        print("*" * (2*lines+1-2*line)+"\n")

    for line in range(lines-1, 0, -1):
        print(" " * (line - 1))
        print("*" * (2 * lines + 1 - 2 * line) + "\n")
```

- **Lab01Test.py**



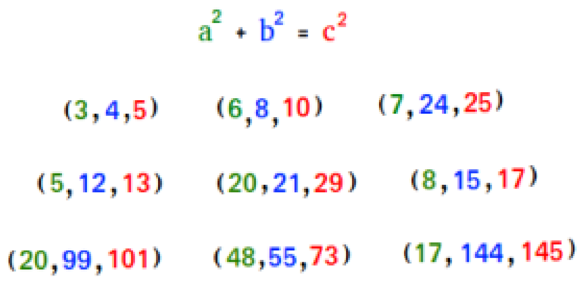
```python
n2 = int(input("drawTriangles(n2) : "))
f1.drawTriangles(n2)
```

[result]



```
drawTriangles(n2) : 5
*********
 *******
  *****
   ***
    *
   ***
  *****
 *******
*********
drawTriangles(n2) : 1
*
```

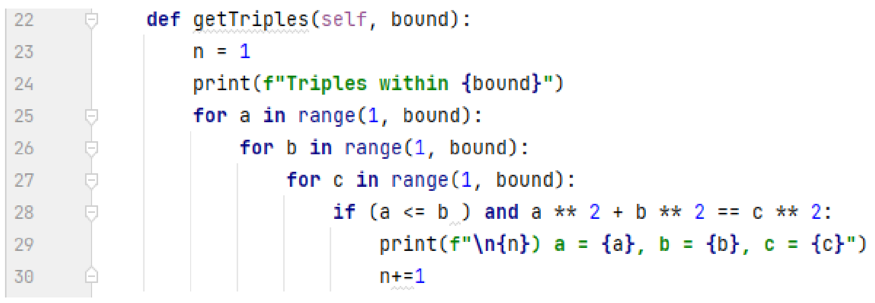
(c) getTriples: A right triangle can have sides that are all integers. The set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for side1, side2 and hypotenuse all no larger than 50.



Pythagorean Triples

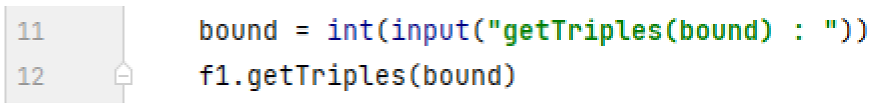
[code]

- **Functions.py**



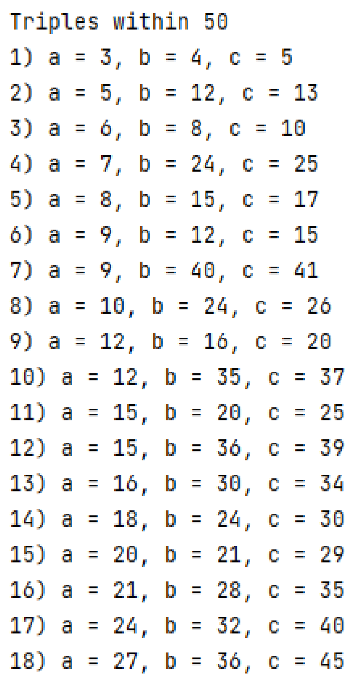
```python
    def getTriples(self, bound):
        n = 1
        print(f"Triples within {bound}")
        for a in range(1, bound):
            for b in range(1, bound):
                for c in range(1, bound):
                    if (a <= b ) and a ** 2 + b ** 2 == c ** 2:
                        print(f"\n{n}) a = {a}, b = {b}, c = {c}")
                        n+=1
```

- **Lab01Test.py**



```python
    bound = int(input("getTriples(bound) : "))
    f1.getTriples(bound)
```

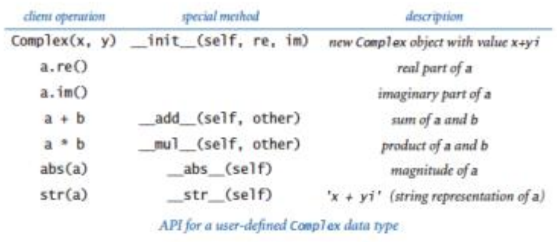
[result]



```
Triples within 50
1) a = 3, b = 4, c = 5
2) a = 5, b = 12, c = 13
3) a = 6, b = 8, c = 10
4) a = 7, b = 24, c = 25
5) a = 8, b = 15, c = 17
6) a = 9, b = 12, c = 15
7) a = 9, b = 40, c = 41
8) a = 10, b = 24, c = 26
9) a = 12, b = 16, c = 20
10) a = 12, b = 35, c = 37
11) a = 15, b = 20, c = 25
12) a = 15, b = 36, c = 39
13) a = 16, b = 30, c = 34
14) a = 18, b = 24, c = 30
15) a = 20, b = 21, c = 29
16) a = 21, b = 28, c = 35
17) a = 24, b = 32, c = 40
18) a = 27, b = 36, c = 45
```

2. Complex Numbers Class

- A complex number is a number of the form x + yi, where x and y are real numbers and i is the square root of -1. The number x is known as the real part of the complex number, and the number y is known as the imaginary part.
- The operations on complex numbers that are needed for basic computations
  - *Addition*: $(x+yi) + (v+wi) = (x+v) + (y+w)i$
  - *Multiplication*: $(x + yi) * (v + wi) = (xv - yw) + (yv + xw)i$
  - *Magnitude*: $|x + yi| = (x^2 + y^2)^{1/2}$
  - *Real part*: $\mathrm{Re}(x + yi) = x$
  - *Imaginary part*: $\mathrm{Im}(x + yi) = y$

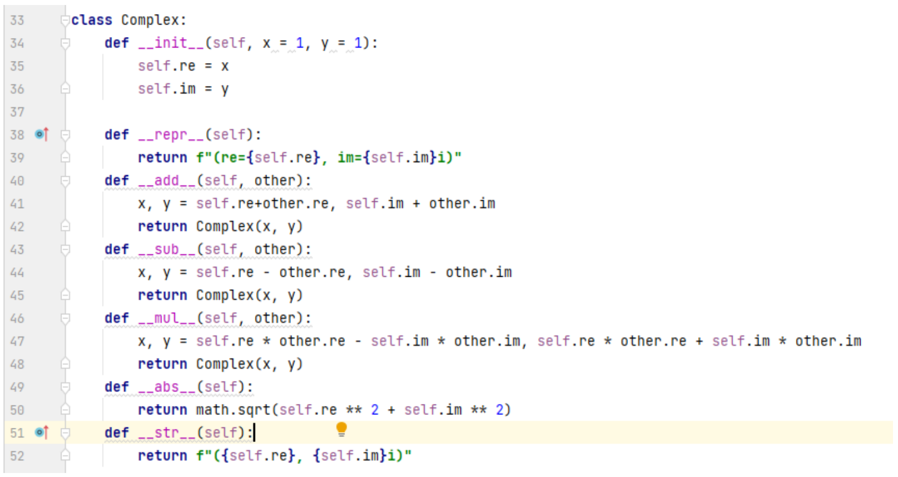


client operation	*special method*	*description*
`Complex(x, y)`	`__init__(self, re, im)`	*new* `Complex` *object with value x+yi*
`a.re()`		*real part of a*
`a.im()`		*imaginary part of a*
`a + b`	`__add__(self, other)`	*sum of a and b*
`a * b`	`__mul__(self, other)`	*product of a and b*
`abs(a)`	`__abs__(self)`	*magnitude of a*
`str(a)`	`__str__(self)`	`'x + yi'` *(string representation of a)*

API for a user-defined `Complex` *data type*

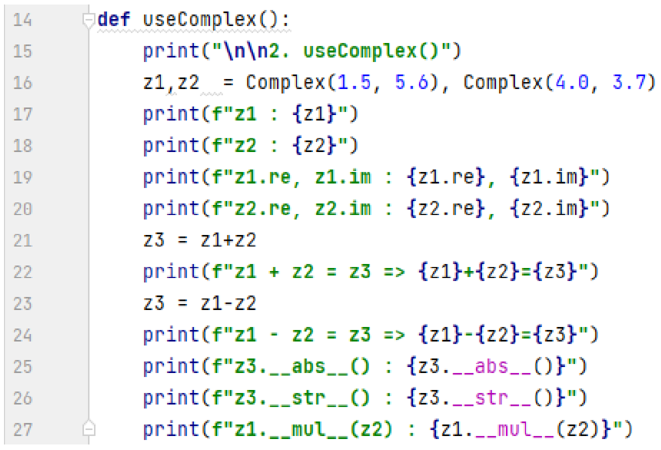
[code]

- **Functions.py**



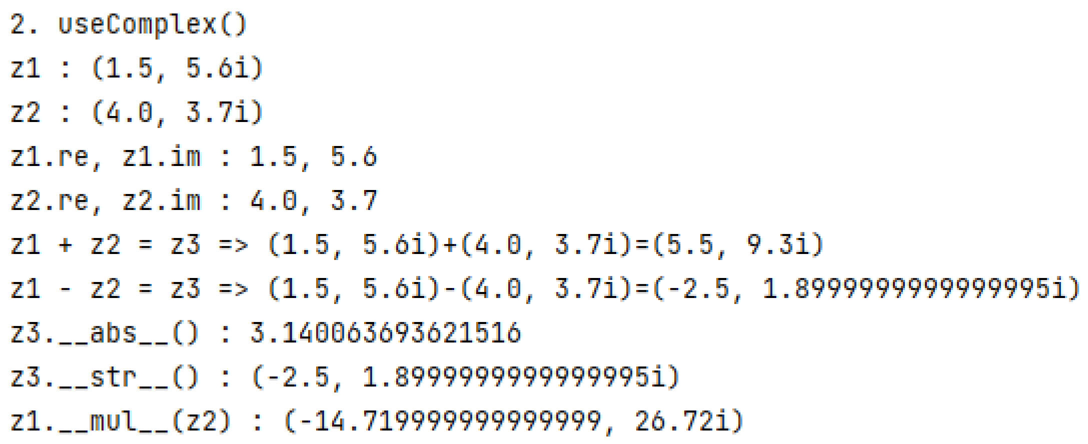
```python
class Complex:
    def __init__(self, x = 1, y = 1):
        self.re = x
        self.im = y

    def __repr__(self):
        return f"(re={self.re}, im={self.im}i)"
    def __add__(self, other):
        x, y = self.re+other.re, self.im + other.im
        return Complex(x, y)
    def __sub__(self, other):
        x, y = self.re - other.re, self.im - other.im
        return Complex(x, y)
    def __mul__(self, other):
        x, y = self.re * other.re - self.im * other.im, self.re * other.re + self.im * other.im
        return Complex(x, y)
    def __abs__(self):
        return math.sqrt(self.re ** 2 + self.im ** 2)
    def __str__(self):
        return f"({self.re}, {self.im}i)"
```

- **Lab01Test.py**



```python
def useComplex():
    print("\n\n2. useComplex()")
    z1,z2  = Complex(1.5, 5.6), Complex(4.0, 3.7)
    print(f"z1 : {z1}")
    print(f"z2 : {z2}")
    print(f"z1.re, z1.im : {z1.re}, {z1.im}")
    print(f"z2.re, z2.im : {z2.re}, {z2.im}")
    z3 = z1+z2
    print(f"z1 + z2 = z3 => {z1}+{z2}={z3}")
    z3 = z1-z2
    print(f"z1 - z2 = z3 => {z1}-{z2}={z3}")
    print(f"z3.__abs__() : {z3.__abs__()}")
    print(f"z3.__str__() : {z3.__str__()}")
    print(f"z1.__mul__(z2) : {z1.__mul__(z2)}")
```

[result]



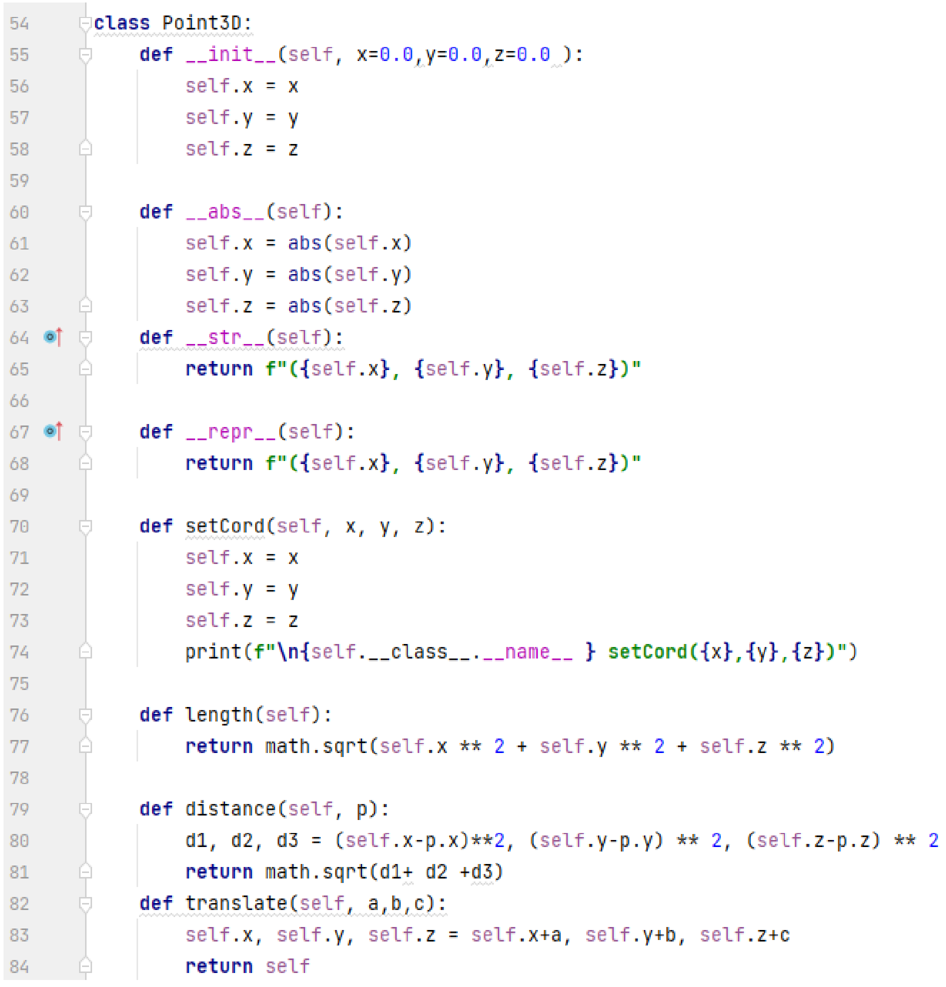
```
2. useComplex()
z1 : (1.5, 5.6i)
z2 : (4.0, 3.7i)
z1.re, z1.im : 1.5, 5.6
z2.re, z2.im : 4.0, 3.7
z1 + z2 = z3 => (1.5, 5.6i)+(4.0, 3.7i)=(5.5, 9.3i)
z1 - z2 = z3 => (1.5, 5.6i)-(4.0, 3.7i)=(-2.5, 1.8999999999999995i)
z3.__abs__() : 3.140063693621516
z3.__str__() : (-2.5, 1.8999999999999995i)
z1.__mul__(z2) : (-14.719999999999999, 26.72i)
```

3. Point Class

- Create a Point3D class and test it in the Lab01Test class. Point3D class should contain the following data and function members
- Data members
  x, y, z, 3D point coordinates of type double
- Function members
  –init–()
  –str–()
  –repr–()
  setCord(double, double, double) : assigns the values for three coordinates
  length() : returns distance between point P and origin (0,0,0)
  distance(Point3D, Point3D): returns distance between two points
  translate(double, double, double): translates (moves) the point in the given directions

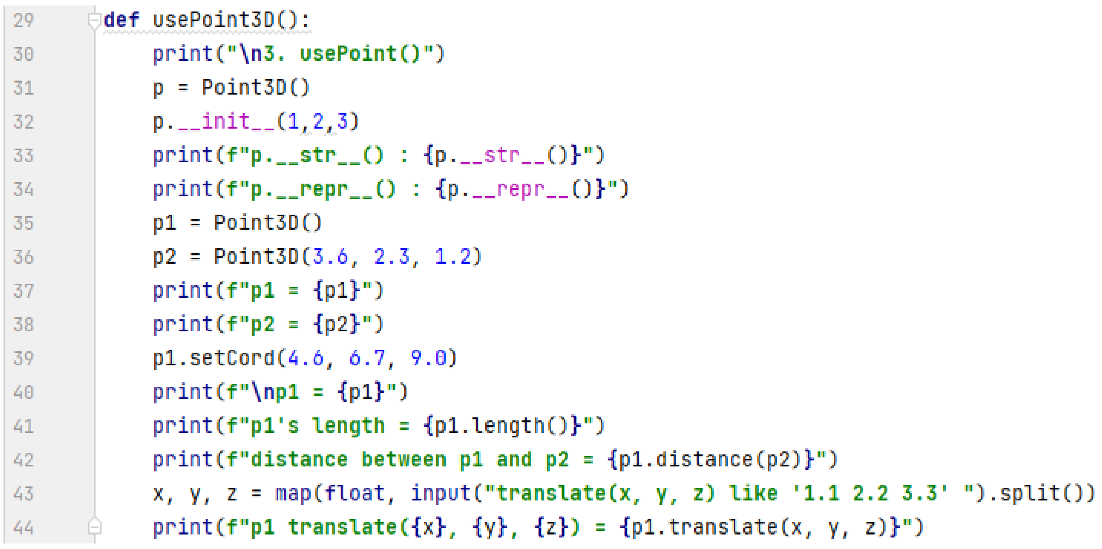
[code]

- **Functions.py**



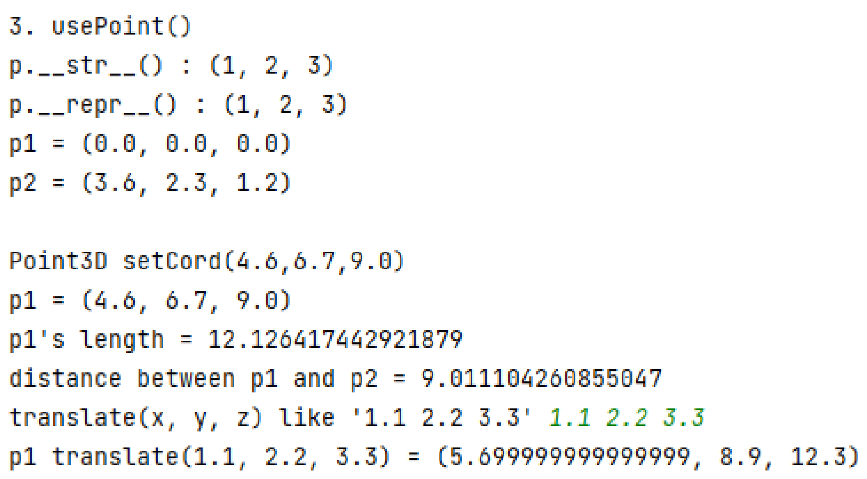
```python
class Point3D:
    def __init__(self, x=0.0,y=0.0,z=0.0 ):
        self.x = x
        self.y = y
        self.z = z

    def __abs__(self):
        self.x = abs(self.x)
        self.y = abs(self.y)
        self.z = abs(self.z)
    def __str__(self):
        return f"({self.x}, {self.y}, {self.z})"

    def __repr__(self):
        return f"({self.x}, {self.y}, {self.z})"

    def setCord(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        print(f"\n{self.__class__.__name__ } setCord({x},{y},{z})")

    def length(self):
        return math.sqrt(self.x ** 2 + self.y ** 2 + self.z ** 2)

    def distance(self, p):
        d1, d2, d3 = (self.x-p.x)**2, (self.y-p.y) ** 2, (self.z-p.z) ** 2
        return math.sqrt(d1+ d2 +d3)
    def translate(self, a,b,c):
        self.x, self.y, self.z = self.x+a, self.y+b, self.z+c
        return self
```

- **Lab01Test.py**



```python
def usePoint3D():
    print("\n3. usePoint()")
    p = Point3D()
    p.__init__(1,2,3)
    print(f"p.__str__() : {p.__str__()}")
    print(f"p.__repr__() : {p.__repr__()}")
    p1 = Point3D()
    p2 = Point3D(3.6, 2.3, 1.2)
    print(f"p1 = {p1}")
    print(f"p2 = {p2}")
    p1.setCord(4.6, 6.7, 9.0)
    print(f"\np1 = {p1}")
    print(f"p1's length = {p1.length()}")
    print(f"distance between p1 and p2 = {p1.distance(p2)}")
    x, y, z = map(float, input("translate(x, y, z) like '1.1 2.2 3.3' ").split())
    print(f"p1 translate({x}, {y}, {z}) = {p1.translate(x, y, z)}")
```

[result]



```
3. usePoint()
p.__str__() : (1, 2, 3)
p.__repr__() : (1, 2, 3)
p1 = (0.0, 0.0, 0.0)
p2 = (3.6, 2.3, 1.2)

Point3D setCord(4.6,6.7,9.0)
p1 = (4.6, 6.7, 9.0)
p1's length = 12.126417442921879
distance between p1 and p2 = 9.011104260855047
translate(x, y, z) like '1.1 2.2 3.3' 1.1 2.2 3.3
p1 translate(1.1, 2.2, 3.3) = (5.699999999999999, 8.9, 12.3)
```