numbers

March 13, 2021

0.1 Adding

[14]:
```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"


    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __radd__(self, other):
        if isinstance(other, (float, int)):
            return Point(self.x, other, self.y + other)
        else:
            return self.__add__(other)

    def __iadd__(self, other):
        if isinstance(other, (float, int)):
            self.x += other
            self.y += other
        else:
            self.x += other.x
            self.y += other.y
        return self


p1 = Point(0, 0)
p2 = Point(1, 3)
p3 = Point(-2, -4)

p2 + p3
print(sum([p1, p2, p2], Point(0, 0)))
p1 += p2
```

```
<Point (x=2, y=6)>
```

0.2 Subtracting

[19]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"

    def __sub__(self, other):
        if not isinstance(other, (Point, int, float)):
              raise TypeError(f"Subtraction is not supported for a Point and␣
↪{other}")
        if isinstance(other, (int, float)):
            return Point(self.x - other, self.y - other)
        else:
            return Point(self.x - other.x, self.y - other.y)

    # if you want to do reverse calculation
    def __rsub__(self, other):
        if not isinstance(other, Point):
            raise TypeError("Try the reverse order. Be careful difference may␣
↪change")
        return self.__sub__(other)

    # if you want to do inplace calculation
    def __isub__(self, other):
        print('in place')
        return self.__sub__(other)

p1 = Point(0, 0)
p2 = Point(1, 3)
p3 = Point(-2, -4)

p2 - p1
p2 -= 5
```

```
in place
```

0.3 Multiplying

- instresting behavioris when you do `5 * 'abc'`, you except failure but because of `rsub/radd` type behavior, it will try `'abc' * 5` and that is valid and would work

[25]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"

    def __mul__(self, other):
        if isinstance(other, (int, float)):
            return Point(self.x * other, self.y * other)
        return Point(self.x * other.x, self.y * other.y)

    def __rmul__(self, other):
        return self.__mul__(other)

    def __imul__(self, other):
        return self.__mul__(other)

p1 = Point(0, 0)
p2 = Point(1, 3)
p3 = Point(-2, -4)

p2 * p3
```

[25]: <Point (x=-2, y=-12)>

0.4 Dividing

- `divmod(5, 4)` returns a tuple with the `div` and the `modulus`

[40]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"

    def __truediv__(self, other):
        if isinstance(other, (int, float)):
            return Point(self.x / other, self.y / other)
        return Point(self.x / other.x, self.y / other.y)

    def __rtruediv__(self, other):
        return self.__truediv__(other)

    def __itruedive__(self, other):
        return self.__truediv__(other)
```

```
    def __floordiv__(self, other):
        if isinstance(other, (int, float)):
            return Point(self.x // other , self.y // other)
        return Point(self.x // other.x, self.y // other.y)

    def __rfloordiv__(self, other):
        return self.__floordiv__(other)

    def __ifloordiv__(self, other):
        return self.__floordiv__(other)


p1 = Point(0, 0)
p2 = Point(1, 3)
p3 = Point(-2, -4)

print(p2 / p3)
p2 //= p3
print(p2)
```

```
<Point (x=-0.5, y=-0.75)>
<Point (x=-1, y=-1)>
```

0.5 Modulo and Powers

- mod is used defined for number and string formatting

[6]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"

    def __mod__(self, other):
        return Point(self.x % other.x, self.y % other.y)

    def __rmod__(self, other):
        print('rmod')

    # mutate object iself or mutate something and then return that
    def __imod__(self, other):
        self.x %= other.x
        self.y %= other.y
        return self

    def __pow__(self, other):
```

```python
        return Point(self.x ** other.x, self.y ** other.y)

    def __rpow__(self, other):
        return other ** self.x

    def __ipow__(self):
        self.x **= other.x
        self.y **= other.y
        return self

p1 = Point(0, 0)
p2 = Point(1, 3)
p3 = Point(-2, -4)

p2 % p3
```

[6]:
```
<Point (x=-1, y=-1)>
```

0.6 Bit Shifting Operations

[15]:
```python
class Binary:
    def __init__(self, number):
        self.number = number
        self._binnumber = bin(number)

    def __repr__(self):
        return f'<Binary number={self.number} binnumber={self._binnumber}>'

    def __lshift__(self, other):
        return Binary(self.number << other.number)

    def __rshift__(self, other):
        return Binary(self.number >> other.number)

    def __rlshift__(self, other):
        return Binary(self .number << other.number)

    def __rrshift__(self, other):
        if isinstance(other, int):
            return Binary(self.number >> other)
        return Binary(self.number >> other.number)

    def __irshift__(self, other):
        self.number >>= other.number
        self._binnumber = bin(self.number)
        return self
```

```python
    def __ilshift__(self, other):
        self.number <<= other.number
        self._binnumber = bin(self.number)
        return self


b1 = Binary(2)
b2 = Binary(32)

b1 << b2

1 >> b1
```

[15]: <Binary number=1 binnumber=0b1>

0.7 Bitwise Logical Operations

- you can use bitwise operators to get bitwise `or-ing`/`and-ing`
- `bin(5 & 7)`

[19]:
```python
class Binary:
    def __init__(self, number):
        self.number = number
        self._binnumber = bin(number)

    def __repr__(self):
        return f'<Binary number={self.number} binnumber={self._binnumber}>'

    def __and__(self, other):
        return Binary(self.number & other.number)

    def __or__(self, other):
        return Binary(self.number | other.number)

    def __xor__(self, other):
        return Binary(self.number ^ other.number)

    def __rand__(self, other):
        if isinstance(other, int):
            return Binary(self.number & other)
        return Binary(self.number & other.number)

    def __ror__(self, other):
        if isinstance(other, int):
            return Binary(self.number | other)
        return Binary(self.number | other.number)
```

```python
    def __rxor__(self, other):
        if isinstance(other, int):
            return Binary(self.number ^ other)
        return Binary(self.number ^ other.number)

    def __iand__(self, other):
        temp = self.__and__(other)
        self.number = temp.number
        self._binnumber = temp._binnumber
        return self

    def __ior__(self, other):
        temp = self.__or__(other)
        self.number = temp.number
        self._binnumber = temp._binnumber
        return self

    def __ixor__(self, other):
        temp = self.__xor__(other)
        self.number = temp.number
        self._binnumber = temp._binnumber
        return self

b1 = Binary(5)
b2 = Binary(7)

b1 & b2
```

[19]:
```
<Binary number=5 binnumber=0b101>
```

0.8 Negative and Postive

- we are going to be trying to deal with things like these `--1/+4/+-1`
- we can do `fraction` addition or subtractions

[25]:
```python
from fractions import Fraction

x = Fraction(1, 4)
x.numerator

print('x')
print(x)
print("")
print("+x")
print(+x)
print("")
```

```
print("-x")
print(-x)
```

```
x
1/4

+x
1/4

-x
-1/4
```

[30]:
```
from datetime import datetime

class Date(datetime):
    def __pos__(self):
        return self.timestamp()

    def __neg__(self):
        return -self.timestamp()


class JString(str):
    def __pos__(self):
        return float(self)

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"

    def __pos__(self):
        return self

    def __neg__(self):
        return Point(-self.x, -self.y)

dt = Date.now()

+dt
-dt

js = JString('1.234')
+js
```

[30]: 1.234

0.9 Absolute Values and Inverse

- ~ is also called inverse
- used in loops to acces other end
- ~ means bounce to the other side -1
- whats happening is its taking binary and then flips it

[45]:
```
class Fraction:
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom

    def __repr__(self):
        return f'<Fraction ({self.num}/{self.denom})'

    def __mul__(self, other):
        return Fraction(self.num * other.num, self.denom * other.denom)

    def __invert__(self):
        return Fraction(self.denom, self.num)


class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"

    # you can use this to return a number an not just point object
    def __abs__(self):
        return Point(abs(self.x), abs(self.y))

    def __invert__(self):
        return Point(self.y, self.x)

p1 = Point(1, 2)
p2 = Point(3, 2)
p3 = Point(-5, -4)

abs(p1)
~p1

f = Fraction(1, 2)
```

```
~f
f * ~f
```

[45]: <Fraction (2/2)

0.10 Integers and Floats

- integers are whole numbers
- floats are anything with decimals
- what if you want an interger representation of a string

[2]:
```
s = '1.23'
print(float(s))

f = 1.23
print(int(f))
```

```
1.23
1
```

[9]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"

    def __int__(self):
        distance = (self.x **2 + self.y **2) ** 0.5
        return int(distance)

    def __float__(self):
        distance = (self.x **2 + self.y **2) ** 0.5
        return float(distance)


# Bad do not do
class Bacteria:
    def __init__(self, size, color):
        self.size = size
        self.color = color

    def __float__(self):
        return float(self.size)

p1 = Point(3, 5)
int(p1)
```

```
float(p1)

bacteria = Bacteria(10.5, "red")
float(bacteria)
```

[9]: 10.5

0.11 Rounding

- be careful about rounding when you have whole numbers verus when you have decimals
- so `3.5` rounding is different than `3.15`
- round can take negative numbers and it basically goes and rounds to the right

[15]:
```
i = 3
f = 3.214

print('round(i)')
print(round(i))
print("")
print('round(f, 2)')
print(round(f, 2))
```

```
round(i)
3

round(f, 2)
3.21
```

[14]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"

    def __round__(self, n=0):
        distance = (self.x **2 + self.y **2) ** 0.5
        return round(distance, n)

p1 = Point(3, 5)
print(round(p1))
```

```
6.0
```

0.12 Floor and Ceiling

- celing gives us the next highest number
- floor gives us the next lowest number

- basically celing is rounding up and floor is rounding down
- `truncate` gives us the whole number
- floor and truncate closely related but behave a bit differently depending on situation

- if you define `float`, you can get `floor` and `ceil` for free

[25]:
```python
from math import floor, ceil, trunc

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point (x={self.x}, y={self.y})>"


    def __floor__(self):
        s = self.x + self.y
        return floor(s)

    def __ceil__(self):
        s = self.x + self.y
        return ceil(s)

    def __trunc__(self):
        return Point(trunc(self.x), trunc(self.y))


p = Point(3.6, 2.7)
print('floor(p)')
print(floor(p))
print("")
print('ceil(p)')
print(ceil(p))
print("")
print("trunc(p)")
print(trunc(p))
```

```
floor(p)
6

ceil(p)
7

trunc(p)
<Point (x=3, y=2)>
```

0.13 Indexing

- we can use a custom object with indexing
- it woul dbe intersting if we could index chars: `letters[ 'a':'c']`
- `__index__` has to return an index
- documentation says that you need to define `int` along with index

[28]:
```python
class Character:

    first = 'A'

    def __init__(self, char):
        self.char = char

    def __repr__(self):
        return f'<Character ("{self.char}")>'

    def __index__(self):
        return ord(self.char) - ord(self.__class__.first)

    def __int__(self):
        return ord(self.char) - ord(self.__class__.first)

a = Character('A')
b = Character('B')
p = Character('P')

letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'


letters[a]
```

[28]:
```
'A'
```

0.14 Complex Numbers

- `complex` is a class
- `complex(1)` gives us '(1+0j)

[29]:
```python
print(complex)
print('')
print('complex(1)')
print(complex(1))
print('')
print('complex(1, 1)')
print(complex(1, 1))
print('')
print('complex(1) + complex(2, 3)')
```

```python
print(complex(1) + complex(2, 3))
```

```
<class 'complex'>

complex(1)
(1+0j)

complex(1, 1)
(1+1j)

complex(1) + complex(2, 3)
(3+3j)
```

[41]:
```python
from pprint import pprint
from math import pi, cos, sin


class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __complex__(self):
        return complex(self.x, self.y)


class Voltage:
    def __init__(self, base_voltage, frequency):
        self.base_voltage = base_voltage
        self.afrequency = 2 * pi * frequency

    def __complex__(self):

        # assumption is time = 1
        real = cos(self.afrequency) * self.base_voltage
        imag = sin(self.afrequency) * self.base_voltage
        return complex(real, imag)

    def at(self, t=0):
        real = cos(self.afrequency * t) * self.base_voltage
        imag = sin(self.afrequency * t) * self.base_voltage
        return complex(real, imag)


v = Voltage(120, 60)
print('complex(v)')
print(complex(v))
```

```python
print('')
seconds = [s/len(seconds) for s in seconds]
results = [v.at(s) for s in seconds]
pprint(results)
```

```
complex(v)
(120-5.1740965224206694e-12j)

[(120+0j),
 (120-3.526982781544377e-13j),
 (120-7.053965563088754e-13j),
 (120-4.468699966111794e-12j),
 (120-1.4107931126177507e-12j)]
```