Classes and Object 1

Thomas Schwarz, SJ
Marquette University

Classes and Objects

- Imperative programming manipulates the state of memory
  - Breaks up tasks through procedures and functions
- Object Oriented Programming
  - Creates objects that interact with each other
  - Objects are defined with a user-defined data type

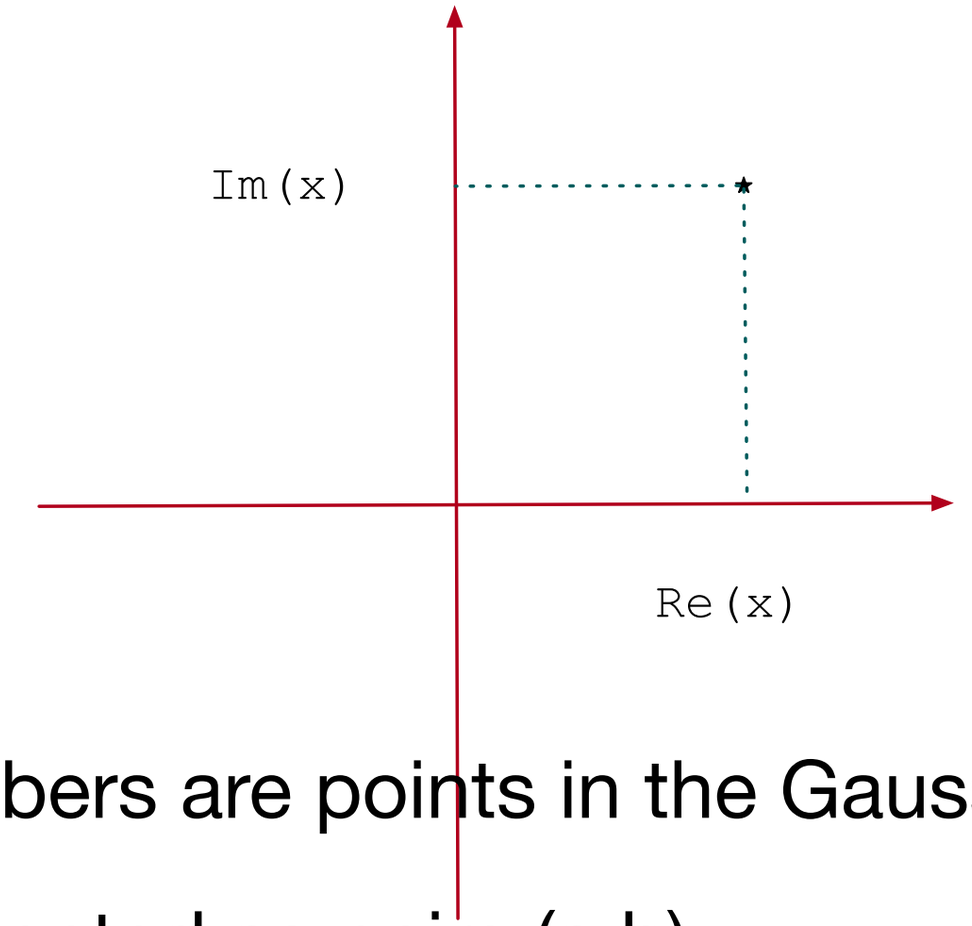
Classes and Objects

- Each object maintains its own state
  - Objects manipulate themselves and other objects through methods

Classes and Objects

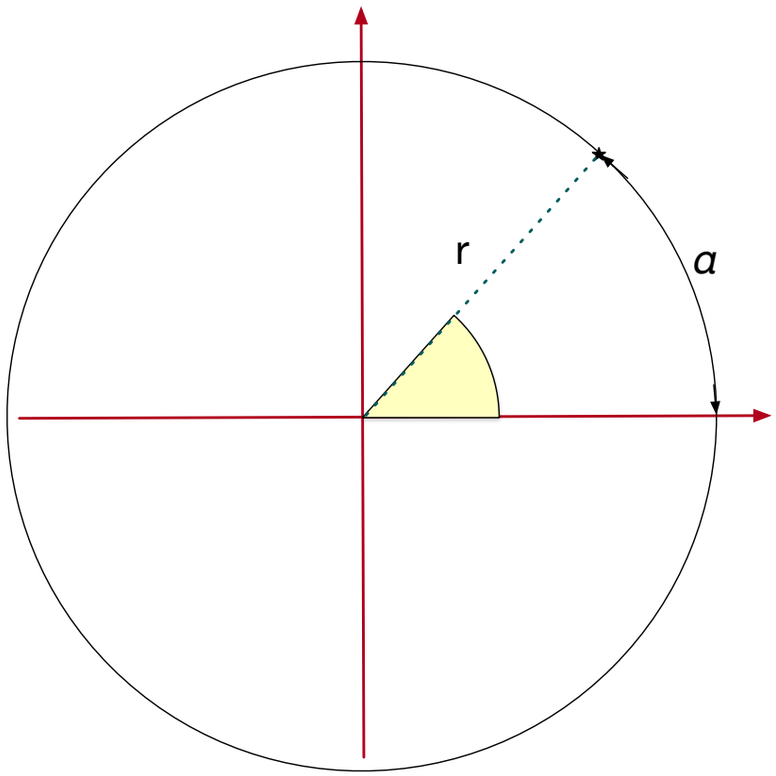
- Running Example:
  - Complex Numbers
  - Complex numbers live in the complex plane
  - Two canonicals way of representing them:
  - Via coordinates (the real and the imaginary part)
  - Via length and angle to x-axis

Complex Numbers



- Complex numbers are points in the Gaussian plane
- Can be represented as pairs (a,b)
  - a is called the real part, b is called the imaginary part
  - Written as `a+ib,` the “algebraic notation”

Complex Numbers



r is the length
α is the phase in radians

- Polar form
  - Given by angle α with x-axis and a length r
  - Written as $r \cdot e^{i\alpha}$

Complex Numbers

- Complex numbers allow various operations such as addition, multiplication, exponentiation,
- The have a length, a real part, and imaginary part, and a phase
- They have a transpose
- And so much more

OOP in Python

- We define the type of an object as a class
  - Objects have fields and methods
    - Fields are like variables
    - Methods are like functions

- A complex number has two fields: the real and the imaginary part
- A method would be the calculation of the length
- Another standard method would be a string describing the number

OOP in Python

- There are two types of fields and methods:
  - Those that belong to the class:
    - Class variables (aka Class fields), Class methods
  - And those that belong to an object
    - Object variables, Object methods

OOP in Python

- To create an object of type class, **“instantiation”**: we define and use an initializer called `__init__()`
- The initializer can have arguments.
- If we create object variables and methods, we use the keyword `self` to refer to the object.

OOP in Python

The double underscore before and after make this a reserved method name.

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

They are called dunder methods.

OOP in Python

The “self” is required. It renders this is an instance method

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

OOP in Python

The real and imaginary are parameters

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

OOP in Python

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

This is the creation of an instance of the class. “self” is hidden, -2 is real and 3 is imaginary

OOP in Python

This defines an instance field called “re”

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

OOP in Python

This defines an instance field called “im”

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

OOP in Python

Assigning self.whatever anywhere in the definition of the class will create an instance object

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

OOP in Python

The two underscores before and after denotes __str__ as a reserved name.

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

OOP in Python

__str__ takes the instance and creates a string. The string should reflect the contents of the object.

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

OOP in Python

When `print` is called on objects of type complex, then Python looks first for a __str__ method and then for a __repr__ method. The __repr__ method is supposed to give more details for debugging.

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        return "{}+i{}".format(self.re, self.im)

if __name__ == "__main__":
    a = Complex(-2, 3)
    b = Complex(1, 2)
    print(a, b)
```

OOP in Python

- Adding Methods
  - Every complex number has a length:

$$|a + bi| = \sqrt{a^2 + b^2}$$

  - To create a method calculating the length:
    - We need one argument: the object (the complex number) itself
    - This argument is called `self`

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        if self.re==0 and self.im==0:
            return "0"
        if self.re==0 and self.im>0:
            return "i{}".format(self.im)
        if self.re==0 and self.im<0:
            return "-i{}".format(-self.im)
        if self.im<0:
            return "{}-i{}".format(self.re, -self.im)
        if self.im>0:
            return "{}+i{}".format(self.re, self.im)
        return str(self.re)
    def length(self):
        return math.sqrt(self.re*self.re+self.im*self.im)


if __name__ == "__main__":
    a = Complex(-2, 3)
    print(a.length())
```

The one argument is self

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        if self.re==0 and self.im==0:
            return "0"
        if self.re==0 and self.im>0:
            return "i{}".format(self.im)
        if self.re==0 and self.im<0:
            return "-i{}".format(-self.im)
        if self.im<0:
            return "{}-i{}".format(self.re, -self.im)
        if self.im>0:
            return "{}+i{}".format(self.re, self.im)
        return str(self.re)
    def length(self):
        return math.sqrt(self.re*self.re+self.im*self.im)


if __name__ == "__main__":
    a = Complex(-2, 3)
    print(a.length())
```

Here is how this function is called: The object is followed by a period

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        if self.re==0 and self.im==0:
            return "0"
        if self.re==0 and self.im>0:
            return "i{}".format(self.im)
        if self.re==0 and self.im<0:
            return "-i{}".format(-self.im)
        if self.im<0:
            return "{}-i{}".format(self.re, -self.im)
        if self.im>0:
            return "{}+i{}".format(self.re, self.im)
        return str(self.re)
    def length(self):
        return math.sqrt(self.re*self.re+self.im*self.im)


if __name__ == "__main__":
    a = Complex(-2, 3)
    print(a.length())
```

So, this method is really a function of one argument, even if it does not look like it.

```
class Complex():
    def __init__(self, real, imaginary):
        self.re = real
        self.im = imaginary
    def __str__(self):
        if self.re==0 and self.im==0:
            return "0"
        if self.re==0 and self.im>0:
            return "i{}".format(self.im)
        if self.re==0 and self.im<0:
            return "-i{}".format(-self.im)
        if self.im<0:
            return "{}-i{}".format(self.re, -self.im)
        if self.im>0:
            return "{}+i{}".format(self.re, self.im)
        return str(self.re)
    def length(self):
        return math.sqrt(self.re*self.re+self.im*self.im)


if __name__ == "__main__":
    a = Complex(-2, 3)
    print(a.length())
```

Here we are referring to an instance field.

Self Test:

Stop the presentation and fire up IDLE

- The phase of a complex number is defined by

$$\textbf{arg}(a+bi) = \begin{cases} \arctan(\frac{b}{a}) & \textbf{if } a > 0 \\ \arctan(\frac{b}{a}) + \pi & \textbf{if } a < 0 \textbf{ and } b \geq 0 \\ \arctan(\frac{b}{a}) - \pi & \textbf{if } a < 0 \textbf{ and } b < 0 \\ \frac{\pi}{2} & \textbf{if } a = 0 \textbf{ and } b > 0 \\ -\frac{\pi}{2} & \textbf{if } a = 0 \textbf{ and } b < 0 \\ \textbf{undefined} & \textbf{if } x = 0 \textbf{ and } b = 0 \end{cases}$$

- Raise a ValueError in the last case, the arctan in the math library is called atan.

```
class Complex():
    def arg(self):
        if self.re == 0 and self.im == 0:
            raise ValueError
        if self.re > 0:
            return math.atan(self.im/self.re)
        if self.re < 0 and self.im >= 0:
            return math.atan(self.im/self.re)+math.pi
        if self.re < 0 and self.im < 0:
            return math.atan(self.im/self.re) - math.pi
        if self.re == 0 and self.im > 0:
            return math.pi/2
        if self.re == 0 and self.im < 0:
            return -math.pi/2

if __name__ == "__main__":
    a = Complex(0, 0)
    print(a.arg())
```

OOP in Python

- Methods can of course return other objects
  - The conjugate of a complex number is obtained by reflecting around the y-axis

$$a + bi = a - bi$$

OOP in Python

```
class Complex():
    def conjugate(self):
        return Complex(self.re, -self.im)
```

```
if __name__ == "__main__":
    a = Complex(2, 3)
    print(a.conjugate())
```

OOP

- Python knows “operator overloading”
  - Instead of adding two complex numbers by saying something like `c=add(a,b)`
  - We can say `c=a+b`

- Python knows a number of such overloads and associates them by reserving function names

```
class Complex():
    def __add__(self, other):
        return Complex(self.re+other.re, self.im+other.im)

if __name__ == "__main__":
    a = Complex(2, 3)
    b = Complex(3, -5)
    print(a+b)
```

We need to return a complex number

Here we are adding two complex numbers.

Self Test

- Overload the subtraction
  - The reserved method name is __sub__
    - (Two underscores before and after)

Solution

```
class Complex():
    def __sub__(self, other):
        return Complex(self.re-other.re, self.im-other.im)

if __name__ == "__main__":
a = Complex(2, 3)
b = Complex(3, -5)
suma = a+b
dif = a-b
print(a, b, suma, dif)
```

OOP in Python

- We can define comparisons via overloading
  - For equality, the reserved method name is __eq__

OOP in Python

- Resolving operators
  - Operator overloading does not have to happen between objects of the same type
  - If there is an expression `a==b`
    - Python first looks into the class definition of a for an __eq__ method that has second parameter the type of b
    - If that fails, Python then looks into the class definition of b for an __eq__method

OOP

- In fact, there is a default comparison between two objects
  - It is based on *IDENTITY* ***NOT*** *EQUALITY*
  - Identity: Two objects are stored at the same location
  - Equality: Two objects have the same fields

```
class Complex():
    def __eq__(self, other):
        if isinstance(other, Complex):
            return self.re==other.re and self.im==other.im
        return NotImplemented
    def __ne__(self, other):
        if isinstance(other,Complex):
            return self.re!=other.re or self.im!=other.im
        return NotImplemented

if __name__ == "__main__":
    a = Complex(2, 3)
    b = Complex(3, -5)
    print(a==b, a!=b)
    print(a==Complex(2,3), a!=Complex(2,3))
```

We only want to compare two complex numbers

```
class Complex():
    def __eq__(self, other):
        if isinstance(other, Complex):
            return self.re==other.re and self.im==other.im
        return NotImplemented
    def __ne__(self, other):
        if isinstance(other,Complex):
            return self.re!=other.re or self.im!=other.im
        return NotImplemented

if __name__ == "__main__":
    a = Complex(2, 3)
    b = Complex(3, -5)
    print(a==b, a!=b)
    print(a==Complex(2,3), a!=Complex(2,3))
```

Two complex numbers are equal if they have the same real and imaginary parts

```
class Complex():
    def __eq__(self, other):
        if isinstance(other, Complex):
            return self.re==other.re and self.im==other.im
        return NotImplemented
    def __ne__(self, other):
        if isinstance(other,Complex):
            return self.re!=other.re or self.im!=other.im
        return NotImplemented

if __name__ == "__main__":
a = Complex(2, 3)
b = Complex(3, -5)
print(a==b, a!=b)
print(a==Complex(2,3), a!=Complex(2,3))
```

If we compare a complex number with a non-complex number then we want to return the constant `NotImplemented`