Abstract Data Type

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Overview

- Abstraction in Programs

- Abstraction Data Type
  - Definition
  - Benefits

- Abstraction Data Type Examples
  - Floating Point Number
  - Complex Number

Abstraction

- The process of isolating implementation details and extracting only **essential property** from an entity

- Program = data + algorithms
- Hence, abstractions in a program:
  - **Data abstraction**
    - What operations are needed by the data
  - **Functional abstraction**
    - What is the purpose of a function (algorithm)

Abstract Data Type (ADT)

- End result of data abstraction
- A collection of ***data*** together with a set of ***operations*** on that data
- **ADT = Data + Operations**

- ADT is a **language independent** concept
  - Different language supports ADT in different ways
  - In Python (OOP Language) the class construct is the best match
- Important Properties of ADT:
  - **Specification:**
    - The supported operations of the ADT
  - **Implementation:**
    - Data structures and actual coding to meet the specification

ADT: Specification and Implementation

- Specification and implementation are disjointed:
  - **One** specification
  - **One or more** implementations
    - **Using different data structure**
    - **Using different algorithm**

- Users of ADT:
  - Aware of the specification **only**
    - Usage only base on the specified operations
  - Do not care / Need not know about the actual implementation
    - i.e. Different implementation do **not** affect the user

Abstraction as Wall : **Illustration**

```python
result = factorial(5)
print(result)
```

User of **`factorial( )`**

```python
def factorial( n ):
    if n == 0:
        return 1

    return n * factorial(n-1)
```

Implementation **1**

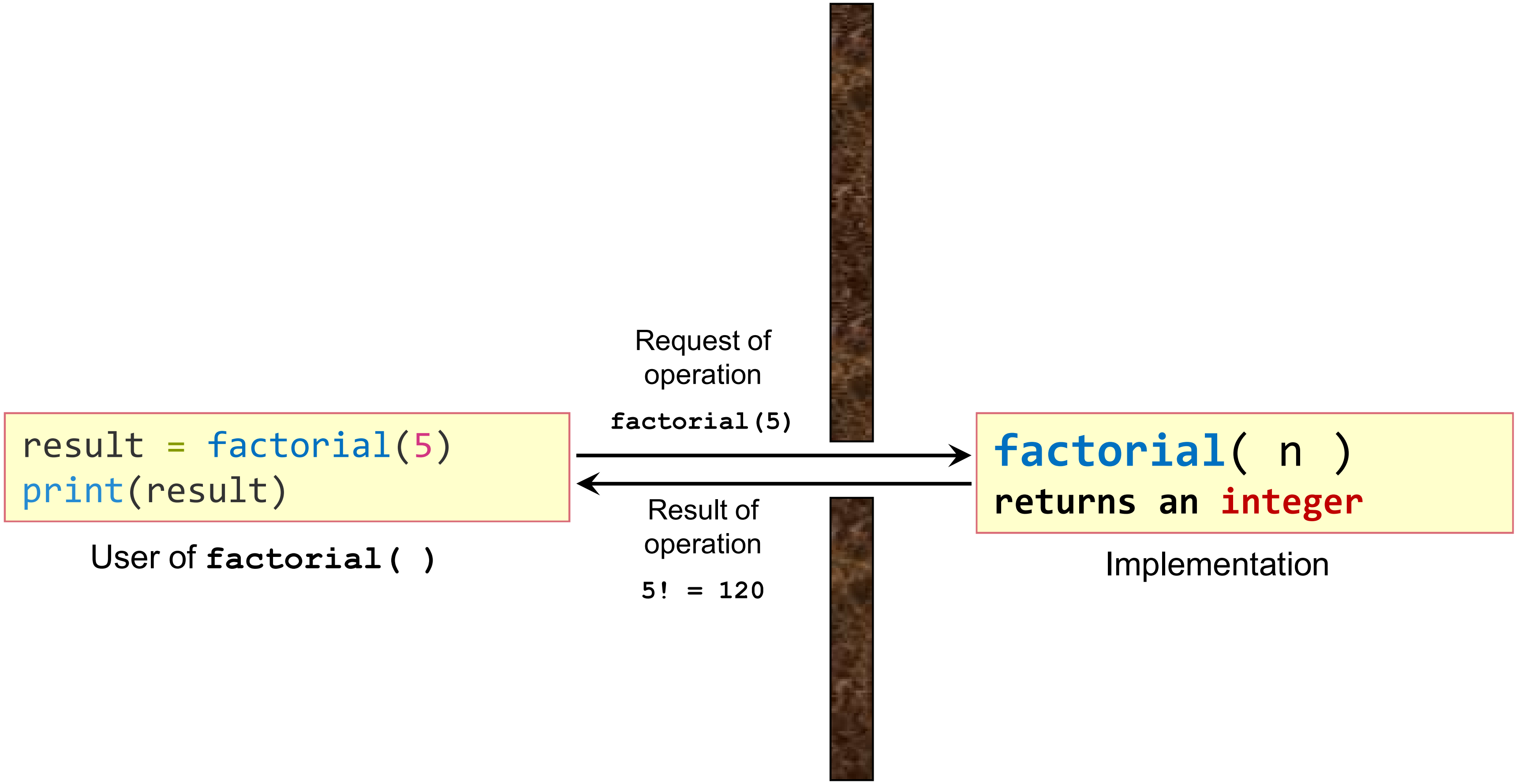
```python
def factorial( n ):
    result = 1

    for i in range(1, n+1):
        result *= i
    return result
```

Implementation **2**

- Users only need to know
  - **`factorial()`**'s purpose
  - Its parameters and return value
- Users **do not** need to know
  - factorial() internal coding
- Different **`factorial()`** coding
  - Does not affect its users!
- We can build a wall to shield **`factorial()`** from **`main()`**!

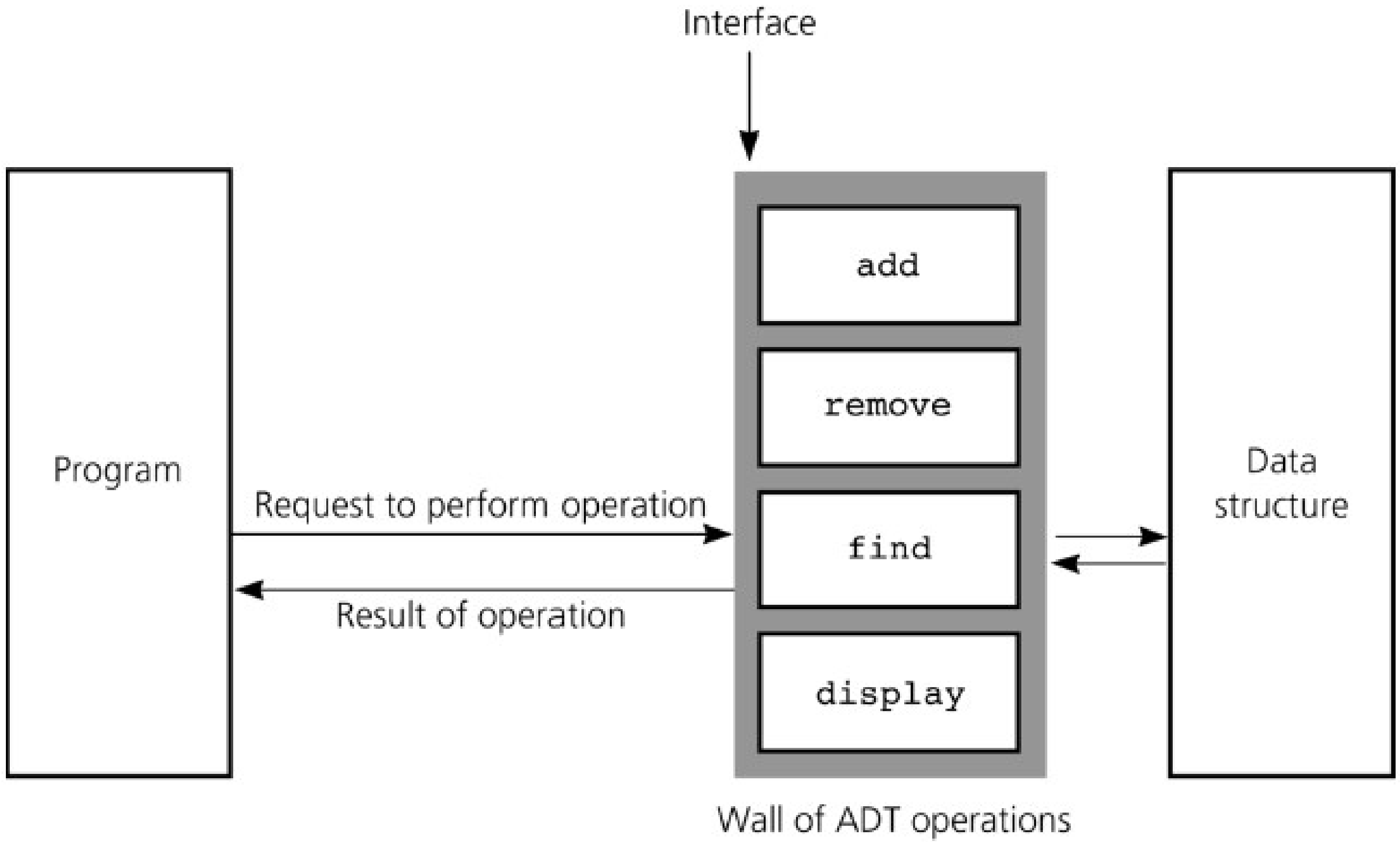
Specification: **Slit in the Wall**



- User only depends on specification
  - Function name, parameters and return type

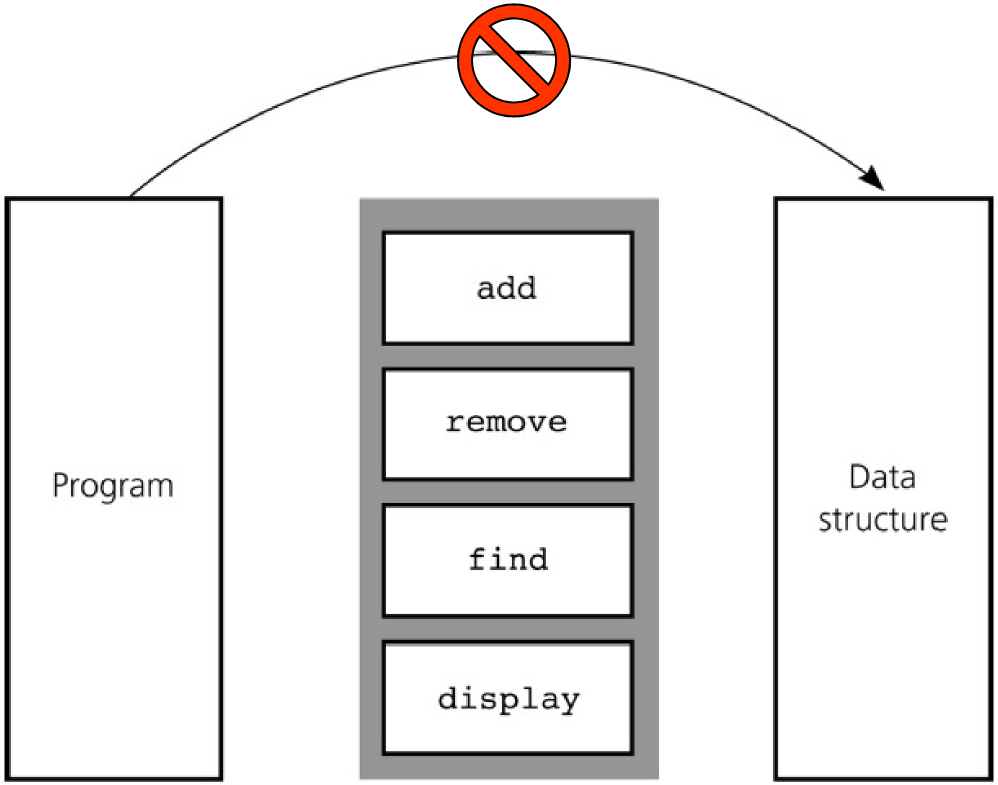
A Wall of ADT operations

- ADT operations provides:
  - Interface to data structure
  - Secure access



Abstraction Violation

- User programs **should not**:
  - Use the underlying data structure directly
  - Depend on implementation details



Wall of ADT operations

Abstract Data Types: When to use?

- When you need to operate on data that are not directly supported by the language
  - E.g. Complex Number, Module Information, Bank Account etc (language dependent!)

- **Simple Steps:**

1. **Design** an abstract data type
2. Carefully **specify all operations** needed
   - Ignore/delay any implementation related issues
3. **Implement** them

Abstract Data Types: **Advantages**

- Hide the unnecessary details by building walls around the data and operations
  - So that changes in either will not affect other program components that use them
- Functionalities are less likely to change
- Localise rather than globalise changes
- Help manage software complexity
- Easier software maintenance

Abstract Data Types: Examples

1. **Primitive Types** as ADTs
   - A simple example

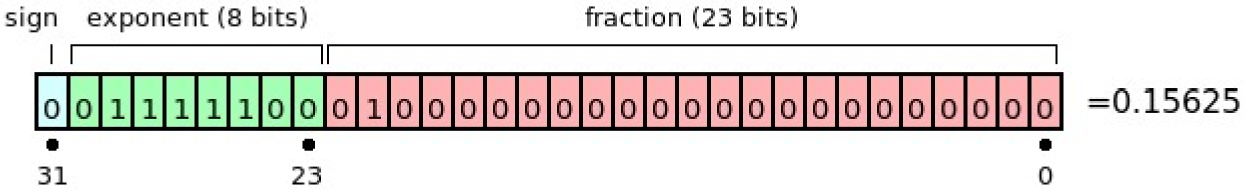
2. **Complex Number** ADT
   - A detailed example to highlight the advantages of ADT

- All data structures covered later in the course are presented as ADTs
  - **Specification**: Essential operations
  - **Implementation**: Actual data structure and coding

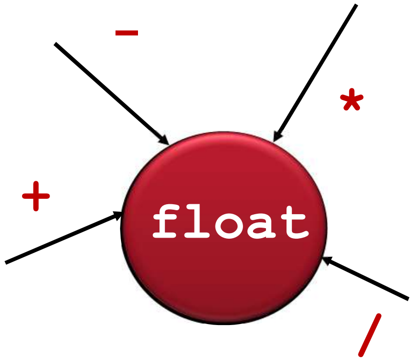
ADT 1 : **Primitive Data Types**

- Predefined data types are examples of ADT
  - E.g. integer, floating point, string, etc

- Representation details are hidden to aid *portability*
  - E.g. float is usually implemented as

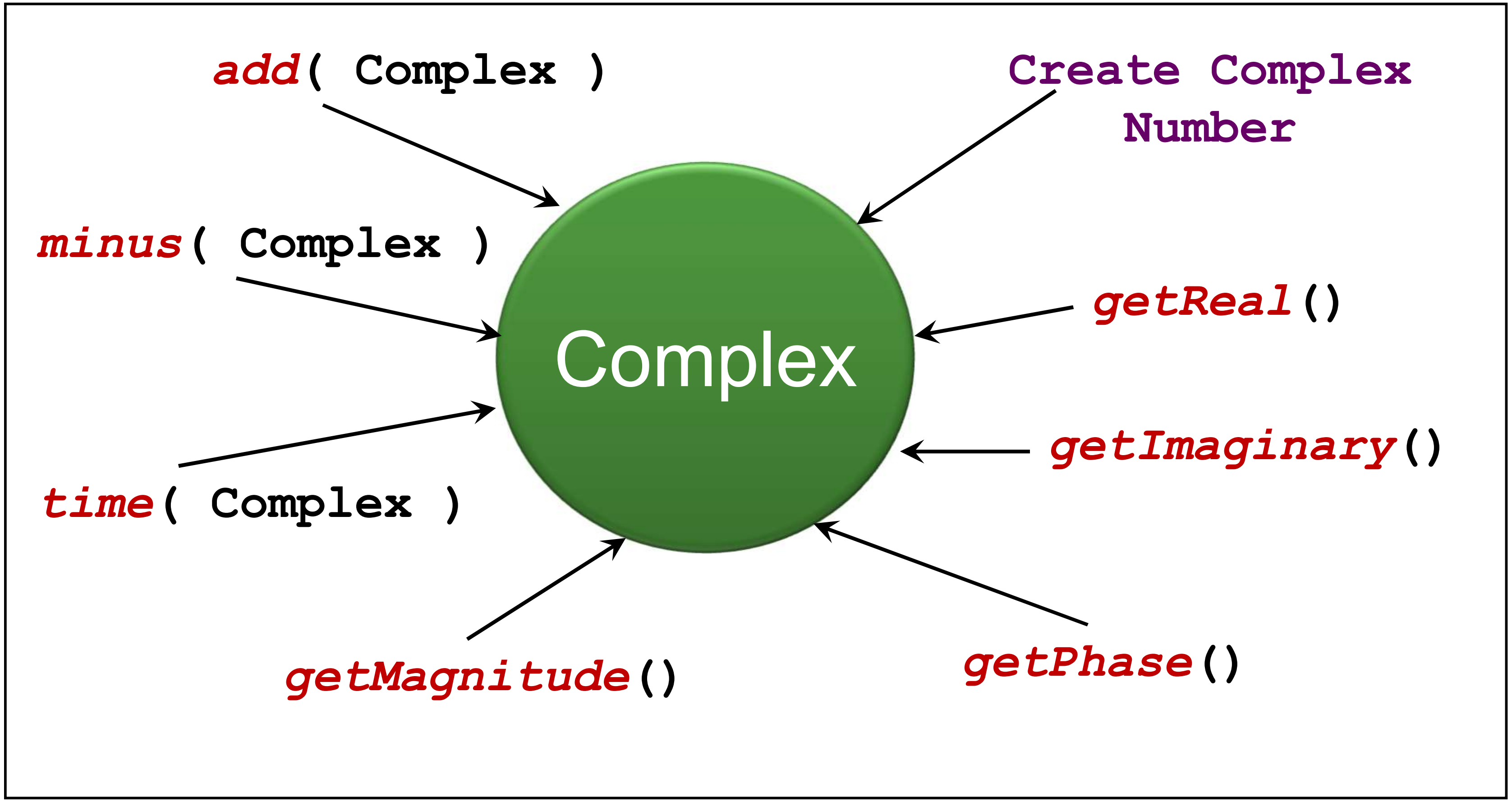


- However, as a user, you don't need to know the above to use float variable in your program



The `float` ADT

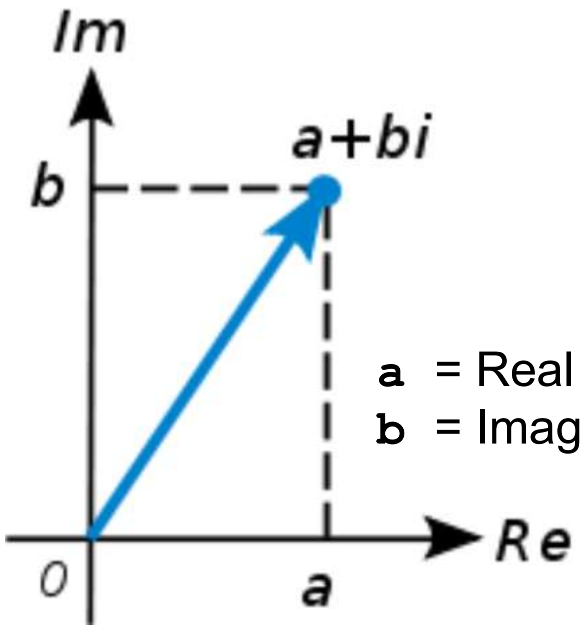
ADT 2 : Complex Number



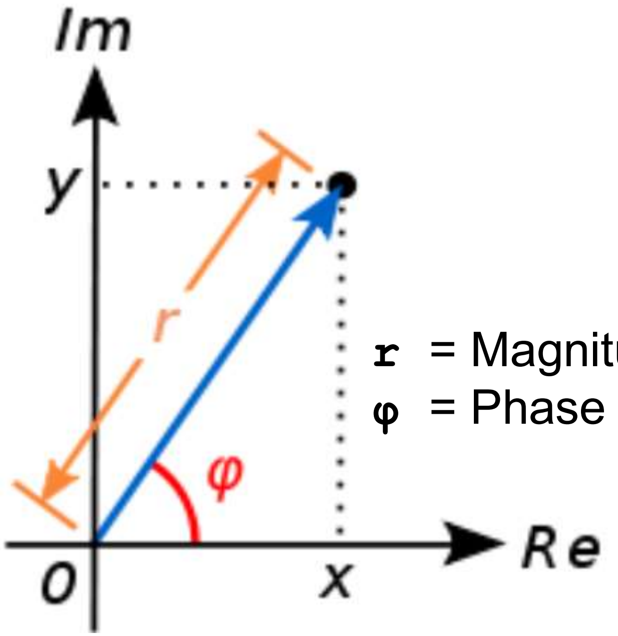
The **Complex** ADT

Complex Number: Representations

- Common representations of complex number:



a = Real Part
b = Imaginary Part



r = Magnitude
φ = Phase (angle)

Rectangular Form

`( a + bi )`

Polar Form

`r( cos φ+ isin φ )`

- Each form is easier to use in certain operations

Complex Number: **Overview**

- **Specification:**
  - Define the common expected operations for a complex number object
- **Implementation:**
  - Complex number can be implemented by at least two different internal representations
    - Keep the ***Rectangular form*** internally OR
    - Keep the ***Polar form*** internally
- Observes the ADT principle in action!

Complex Number: Design

- Complex number can be implemented as two classes:
  - **Each utilize different internal representation**

- A better alternative:
  - Let us define a **abstract base class** which captures the essential operations of a complex number
  - The super class is independent from the actual representation

- We can then utilize:
  - **Inheritance** and **polymorphism** to provide different actual implementations without affecting the user

Abstract Base Class: ComplexBase

Abstract Base Class

```python
#imports not shown

class ComplexBase(ABC):
    @abstractmethod
    def getReal(self):
        pass

    @abstractmethod
    def getImaginary(self):
        pass

    @abstractmethod
    def getMagnitude(self):
        pass

    @abstractmethod
    def getPhase(self):
        pass

    @abstractmethod
    def add(self, other):
        pass

    @abstractmethod
    def minus(self, other):
        pass

    @abstractmethod
    def time(self, another):
        pass

    @abstractmethod
    def toRectangularString(self):
        pass

    @abstractmethod
    def toPolarFormString(self):
        pass
```

- **ComplexBase** is a "placeholder" class
  - Specifies all necessary operations but with no actual implementation

ComplexBase.py

User Program Example: **Preliminary**

```
def main():

    c1 =
    c2 =

    print("Complex number c1:")
    print(c1.toRectangularString())
    print(c1.toPolarFormString())

    print("Complex number c2:")
    print(c2.toRectangularString())
    print(c2.toPolarFormString())

    print("add c2 to c1")
    c1.add(c2)

    print("Complex number c1:")
    print(c1.toRectangularString())


if __name__ == "__main__":
        main()
```

To be replaced by actual implementations of the `ComplexBase` class

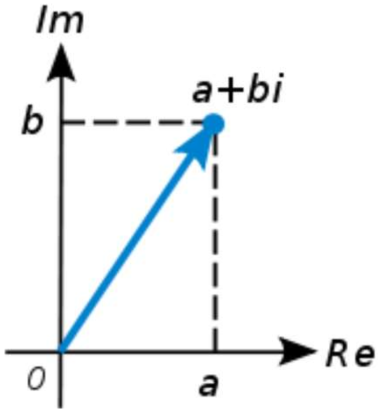
As a user, we can use the methods without worrying about the actual implementation!

`User-ComplexNumber.py`

Rectangular Form Representation

COMPLEX NUMBER

VERSION A



ComplexRectangular: Specification

```
class ComplexRectangular(ComplexBase):

    def __init__(self, real, imag):
        self._real = real
        self._imag = imag

    def getReal(self):
        return self._real

    def getImaginary(self):
        return self._imag

    def getMagnitude(self):
        return math.sqrt( self._real**2 + self._imag**2)
```

The real and imaginary part are kept as object attributes

Methods in this class do not have the **abstract method** decorator
➔ we will give actual implementation

`ComplexRectangular`: Implementation

```
def getPhase(self):
    if self._real > 0 or self._imag != 0:
        den = math.sqrt(self._real**2 + self._imag**2)\
              + self._real
        radian = 2 * math.atan( self._imag / den)
    elif self._real < 0 and self._
        radian = math.pi / 2
    else:
        radian = None
    return radian

def add(self, other):
    self._real = self._real + other.getReal()
    self._imag = self._imag + other.getImaginary()

def minus(self, other ):
    self._real = self._real - other.getReal()
    self._imag = self._imag - other.getImaginary()
```

Algebra of complex numbers:

(i) Addition:
$(a + ib) + (c + id) = (a + c) + i\,(b + d)$

(ii) Subtraction:
$(a + ib) - (c + id) = (a - c) + i\,(b - d)$

(iii) Multiplication:
$(a + ib)\,(c + id) = (ac - bd) + i\,(ad + bc)$

ComplexRectangular: Implementation

```
    def time( self, other ):
        realNew = self._real * other.getReal() \
                  - self._imag * other.getImaginary()
        imagNew  = self._real * other.getImaginary() \
                  + self._imag * other.getReal()
        self._real = realNew
        self._imag = imagNew

    def toRectangularString(self):
        return "({:.3f}, {:.3f}i)".format(self.getReal(),
        self.getImaginary())

    def toPolarFormString(self):
        return "{0:.3f}(cos {1:.3f}, i sin
        {1:.3f})".format(self.getMagnitude(), self.getPhase())
```

- Notes:
  - We chose to avoid more advanced Python syntax (e.g. class property setter / getter, etc)
  - Feel free to experiment after you understood the basic premise

User Program Example: Version 2.0

```
def main():

    c1 = ComplexRectangular(30, 10)
    c2 = ComplexRectangular(20, 20)

    print("Complex number c1:")
    print(c1.toRectangularString())
    print(c1.toPolarFormString())

    print("Complex number c2:")
    print(c2.toRectangularString())
    print(c2.toPolarFormString())

    print("add c2 to c1")
    c1.add(c2)

    print("Complex number c1:")
    print(c1.toRectangularString())
```

c1, **c2** are initialized as **ComplexRectangular** objects

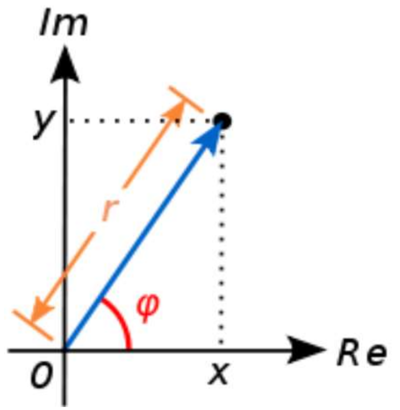
Observe that the implementation details doesn't affect the behavior of an ADT, i.e. user program is unchanged!

User-ComplexNumber.py

Polar Form Representation

COMPLEX NUMBER

VERSION B



ComplexPolar: Implementation

```
class ComplexPolar(ComplexBase):

    def __init__(self, magnitude, phase):
        self._mag = magnitude
        self._phase = phase


    def getReal(self):
        return self._mag * math.cos(self._phase)

    def getImaginary(self):
        return self._mag * math.sin(self._phase)

    def getMagnitude(self):
        return self._mag

    def getPhase(self):
        return self._phase
```

The magnitude and phase from the complex plane origin are kept as object attributes

Note that the two parameters have different meaning compared to the **`ComplexRectangular`** version

Since we keep only magnitude and phase as attributes, the real and imaginary parts need to be calculated

`ComplexPolar.py`

ComplexPolar: Implementation

```python
def _convertPhaseAngle(self, real, imag ):
    if real != 0:
        den = math.sqrt(real**2 + imag**2) + real
        radian = 2 * math.atan( imag / den)
    elif imag > 0:
        radian = math.pi / 2
    else:
        radian = -math.pi / 2
    return radian

def add(self, other):
    real = self.getReal() + other.getReal()
    imag = self.getImaginary() + other.getImaginary()

    self._mag =  math.sqrt( real**2 + imag**2 )
    self._phase = self._convertPhaseAngle(real, imag)

def minus(self, other ):
    real = self.getReal() - other.getReal()
    imag = self.getImaginary() - other.getImaginary()

    self._mag =  math.sqrt( real**2 + imag**2 )
    self._phase = self._convertPhaseAngle(real, imag)
```

An example of "helper method", used internally to simplify coding

Convert to rectangular form for addition

Convert back to polar form

Similar idea for subtraction

`ComplexPolar`: Implementation

```
def time( self, another ):
    self._mag *= another.getMagnitude()
    self._phase += another.getPhase()
```

Multiplication in Polar form is real easy though!

```
def toRectangularString(self):
```

Code similar to `ComplexRectangular`. Not Shown.

```
def toPolarFormString(self):
```

Code similar to `ComplexRectangular`. Not Shown.

- At this point:
  - We have two **independent implementations** of complex number
  - They have different internal working, but support the same behavior

User Program Example: Version 3.0

```
def main():

    c1 = ComplexPolar(31.62, 0.322)
    c2 = ComplexPolar(28.28, 0.785)

    print("Complex number c1:")
    print(c1.toRectangularString())
    print(c1.toPolarFormString())

    print("Complex number c2:")
    print(c2.toRectangularString())
    print(c2.toPolarFormString())

    print("add c2 to c1")
    c1.add(c2)

    print("Complex number c1:")
    print(c1.toRectangularString())


if __name__ == "__main__":
        main()
```

Note that `ComplexPolar` constructs with magnitude and phase

No change to code otherwise

User-ComplexNumber.py

User Program Example: Version 4.0

```python
def main():

    c1 = ComplexRectangular(30, 10)
    c2 = ComplexPolar(28.28, 0.785)

    print("Complex number c1:")
    print(c1.toRectangularString())
    print(c1.toPolarFormString())

    print("Complex number c2:")
    print(c2.toRectangularString())
    print(c2.toPolarFormString())

    print("add c2 to c1")
    c1.add(c2)

    print("Complex number c1:")
    print(c1.toRectangularString())


if __name__ == "__main__":
        main()
```

The **c1** and **c2** need not be the same implementation!

Can you figure out how **c1** and **c2** can interoperate?

Complex Number: **Summary**

- This example highlights:
  - Separation of **specification** and **implementation**
  - A specification can have multiple implementations

- Why is this useful?
  1. We can try out **different strategies** in implementation **without affecting the user**
  2. We can use the best implementation in a certain situation
     - E.g. If multiplication is going to be the most common operations in a complex number program, we can choose to use the `polar form` implementation

Summary

- Abstraction is a powerful technique
  - Data Abstraction
  - Function Abstraction

- Abstract Data Type
  - External Behavior
    - The specification

  - Internal Coding
    - The actual implementations

References

- [Carrano]
  - 4th / 5th Edition, Chapter 3

- [Koffman & Wolfgang]
  - Chapter 1.4

- Source:
  - The two diagrams of complex number representation are taken from http://wikipedia.org