

Lecture 5: Iterative versus Recursive Algorithms - Memoization // Object-Oriented Programming

Chiheb Ben Hammouda

Mathematical Institute, Utrecht University

Python and R Course (WISB153)

May 20, 2025

Plan of Lecture 5

1 Iterative versus Recursive Algorithms - Memoization

2 Object-Oriented Programming

- Motivation and Terminology
- Overloading
- Encapsulation
- Inheritance and Polymorphism

1 Iterative versus Recursive Algorithms - Memoization

2 Object-Oriented Programming

- Motivation and Terminology
- Overloading
- Encapsulation
- Inheritance and Polymorphism

Time complexity for Fibonacci – iterative

Example: computing Fibonacci numbers.

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2.$$

```
def compute_fib_it(n):  
    fib = [0,1]  
    for i in range(2,n):  
        next_fib = fib[i-1]+fib[i-2]  
        fib.append(next_fib)  
    return fib[n-2]+fib[n-1]
```

what is the time complexity?

Time complexity for Fibonacci – iterative

Example: computing Fibonacci numbers.

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2.$$

```
def compute_fib_it(n):  
    fib = [0,1]  
    for i in range(2,n):  
        next_fib = fib[i-1]+fib[i-2]  
        fib.append(next_fib)  
    return fib[n-2]+fib[n-1]
```

what is the time complexity?

- For a list, creating a list with two elements, adding an element to the end of the list and retrieving the last or pre-last element are all $\mathcal{O}(1)$.
- In the for-loop, we repeat $\mathcal{O}(1)$ operations n times.

⇒ So the whole iterative algorithm runs in $\mathcal{O}(n)$.

Recursive Algorithm: Fibonacci

```
def fib_rec(n):  
    if n==0:  
        return 0  
    elif n==1:  
        return 1  
    else:  
        return fib_rec(n-1) + fib_rec(n-2)
```

Question: what is the time complexity? Can we apply the master theorem?

Time Complexity of Recursive Fibonacci Algorithm



$$\begin{aligned}T(n) &= T(n-1) + T(n-2) + 1 \\&\approx 2T(n-1) + 1 = 2(T(n-2) + T(n-3) + 1) + 1 \\&\approx 2^2 T(n-2) + 3 \\&\vdots \\&\approx 2^k T(n-k) + (2^k - 1) \\&\vdots \\&= 2^n T(0) + (2^n - 1) = \mathcal{O}(2^n)\end{aligned}$$

Time Complexity of Recursive Fibonacci Algorithm



$$\begin{aligned}T(n) &= T(n-1) + T(n-2) + 1 \\&\approx 2T(n-1) + 1 = 2(T(n-2) + T(n-3) + 1) + 1 \\&\approx 2^2 T(n-2) + 3 \\&\vdots \\&\approx 2^k T(n-k) + (2^k - 1) \\&\vdots \\&= 2^n T(0) + (2^n - 1) = \mathcal{O}(2^n)\end{aligned}$$

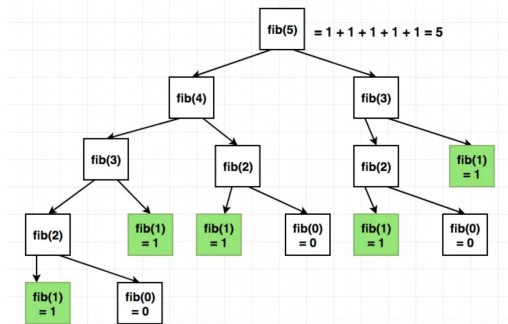
- We can prove that

$$T(n) = \Theta(\phi^n),$$

where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.62$ is the **golden ratio**...

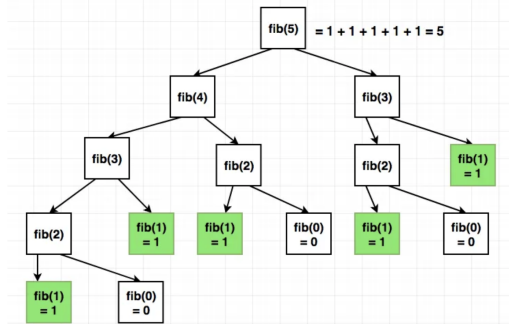
Why is the Recursive Fibonacci Algorithm Inefficient?

- Explosion (every node will split into two subbranches)!



Why is the Recursive Fibonacci Algorithm Inefficient?

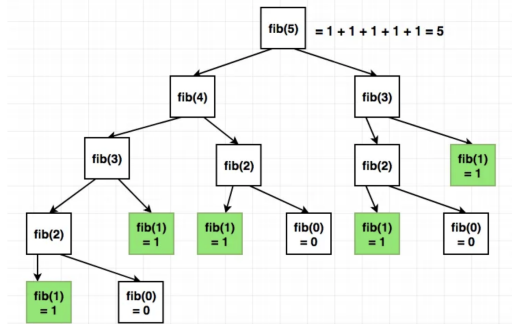
- Explosion (every node will split into two subbranches)!



- Why Inefficient:

Why is the Recursive Fibonacci Algorithm Inefficient?

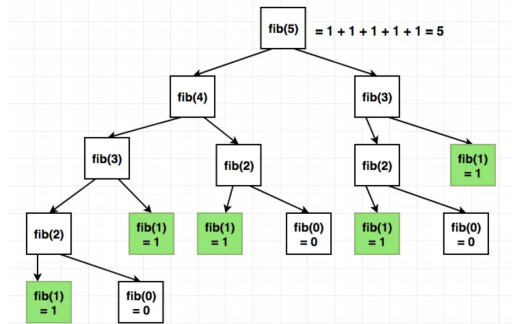
- Explosion (every node will split into two subbranches)!



- Why Inefficient:** we compute f_{n-3} in order to compute f_{n-1} , and then compute f_{n-3} **from scratch again** when computing f_{n-2} .

Why is the Recursive Fibonacci Algorithm Inefficient?

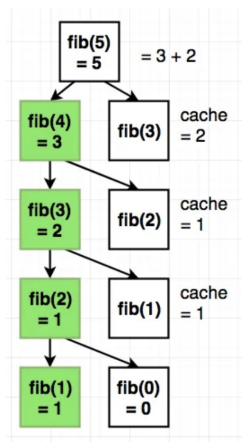
- Explosion (every node will split into two subbranches)!



- **Why Inefficient:** we compute f_{n-3} in order to compute f_{n-1} , and then compute f_{n-3} **from scratch again** when computing f_{n-2} .
- The recursive algorithm can be sped up by avoiding repeated calculations, which leads to [memoization](#).

Memoization

- **Memoization**: store intermediate computation results to avoid computing the same values multiple times (avoid unnecessary function calls) → **speed up**.



Memoization

- **Memoization**: store intermediate computation results to avoid computing the same values multiple times (avoid unnecessary function calls) → **speed up**.
- A recursive Python program with memoization using a **dictionary** named `memo` with key-value pairs:

```
def fib_rec_memo(n, memo = {0:0, 1:1}):  
    if n in memo:  
        fn = memo[n]  
    else:  
        fn = fib_rec_memo(n-1, memo) + fib_rec_memo(n-2, memo)  
        memo[n] = fn  
    return fn
```

What is the time complexity?

Memoization

- **Memoization**: store intermediate computation results to avoid computing the same values multiple times (avoid unnecessary function calls) → **speed up**.
- A recursive Python program with memoization using a **dictionary** named `memo` with key-value pairs:

```
def fib_rec_memo(n, memo = {0:0, 1:1}):  
    if n in memo:  
        fn = memo[n]  
    else:  
        fn = fib_rec_memo(n-1, memo) + fib_rec_memo(n-2, memo)  
        memo[n] = fn  
    return fn
```

- The program checks in $\mathcal{O}(1)$ time whether you have already calculated f_n :
 - If yes, then the result is used;
 - If no, then it is calculated and stored in $\mathcal{O}(1)$ time.

→ **Complexity of $\mathcal{O}(n)$**

Iterative versus Recursive Algorithm: Factorial Example

- A recursive Python program and an iterative program for computing $n!$:

```
import time

def factorial_rec(n):
    if n==0:
        return 1
    else:
        return n*factorial_rec(n-1)

def factorial_it(n):
    f = 1
    for i in range(1,n+1):
        f = f * i
    return f

def test_function(f, desc, n):
    start = time.time_ns()
    f(n)
```


Iterative versus Recursive Algorithm: Factorial Example

- A recursive Python program and an iterative program for computing $n!$:

```
import time

def factorial_rec(n):
    if n==0:
        return 1
    else:
        return n*factorial_rec(n-1)

def factorial_it(n):
    f = 1
    for i in range(1,n+1):
        f = f * i
    return f

def test_function(f, desc, n):
    start = time.time_ns()
    f(n)
```

Time for Recursive and Iterative Algorithm for $n!$

- Output of the test for $n = 100$ in seconds:
time for iterative 100! is 1.2159e-05
time for recursive 100! is 2.0027e-05

Time for Recursive and Iterative Algorithm for $n!$

- Output of the test for $n = 100$ in seconds:
time for iterative 100! is 1.2159e-05
time for recursive 100! is 2.0027e-05
- Output of the test for $n = 1000$ in seconds:
time for iterative 1000! is 0.0002639

```
Traceback (most recent call last):
  File "factorial_test.py", line 20, in <module>
    factorial_rec(n)
  File "factorial_test.py", line 6, in factorial_rec
    return n*factorial_rec(n-1)
  File "factorial_test.py", line 6, in factorial_rec
    return n*factorial_rec(n-1)
[Previous line repeated 996 more times]
  File "factorial_test.py", line 3, in factorial_rec
    if n==0:
RecursionError: maximum recursion depth exceeded
in comparison
```

Recursion Error

- A `RecursionError` in Python is an exception that occurs when the maximum recursion depth is exceeded (**Memory issue**), i.e., the number of times the function calls itself recursively exceeds a limit.

Recursion Error

- A **RecursionError in Python** is an exception that occurs when the maximum recursion depth is exceeded (**Memory issue**), i.e., the number of times the function calls itself recursively exceeds a limit.
- How to fix RecursionError in Python:
 - **Adding a base case:** the most common cause of a recursion error is that the function does not have a base case to stop the recursion.
 - **Increasing the recursion limit:** Python has a default maximum recursion depth of 1000. (**Not advised: can cause a crash**)

```
import sys
sys.setrecursionlimit(2000)
```
- **Using an iterative approach (e.g. a for or while loops) for large n .**

1 Iterative versus Recursive Algorithms - Memoization

2 Object-Oriented Programming

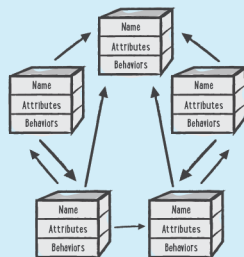
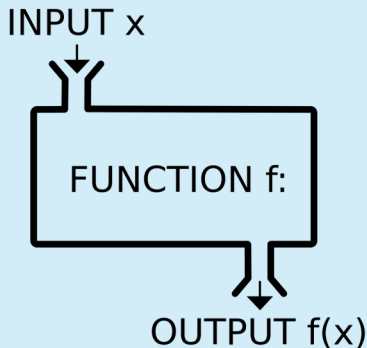
- Motivation and Terminology
- Overloading
- Encapsulation
- Inheritance and Polymorphism

Object-Oriented Programming: Motivation

- Previously, we used functions to introduce structure into programs and enable code reuse.
- With the **object-oriented approach** (OO),

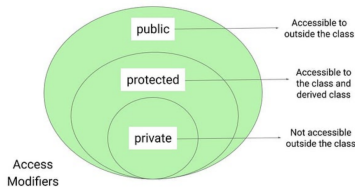
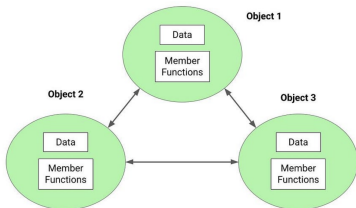
Object-Oriented Programming: Motivation

- Previously, we used functions to introduce structure into programs and enable code reuse.
- With the **object-oriented approach** (OO),
 - We can introduce even more structure.
 - Easier to : (i) re-use code; (ii) understand code; (iii) debug.

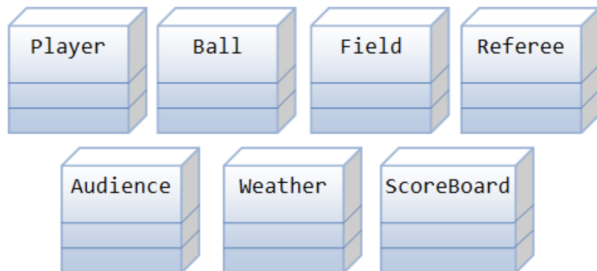


Object-Oriented Programming: Motivation

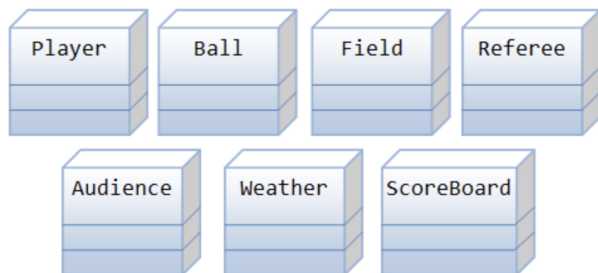
- Previously, we used functions to introduce structure into programs and enable code reuse.
- With the **object-oriented approach (OO)**,
 - **We can introduce even more structure.**
 - Easier to : (i) re-use code; (ii) understand code; (iii) debug.
 - **Hide internal details for specific users (Encapsulation).**
 - We can make changes without the user noticing.
 - We may not want the user to be able to access or view all the data.



OO Example: Classes (Objects) in Computer Soccer Game



OO Example: Classes (Objects) in Computer Soccer Game



- Some classes, like **Ball** and **Audience**, can be reused in another application (Example: Computer basketball game).
- Little or no modification needed for reuse.

Classes in a Soccer Game

Player:

Classes in a Soccer Game

Player:

- **Attributes:** Name, Number, Location, Skill Level, Team, etc.
- **Methods:** Run(), Jump(), KickTheBall(), PassBall(), etc.

Ball:

Classes in a Soccer Game

Player:

- **Attributes:** Name, Number, Location, Skill Level, Team, etc.
- **Methods:** Run(), Jump(), KickTheBall(), PassBall(), etc.

Referee:

Ball:

- **Attributes:** Size, Color, Velocity, Position, etc.
- **Methods:** Move(), Bounce(), Stop(), etc.

Classes in a Soccer Game

Player:

- **Attributes:** Name, Number, Location, Skill Level, Team, etc.
- **Methods:** Run(), Jump(), KickTheBall(), PassBall(), etc.

Referee:

- **Attributes:** Name, Experience, DecisionMakingAbility, etc.
- **Methods:** ShowCard(), MakeDecision(), etc.

Ball:

- **Attributes:** Size, Color, Velocity, Position, etc.
- **Methods:** Move(), Bounce(), Stop(), etc.

Field:

Classes in a Soccer Game

Player:

- **Attributes:** Name, Number, Location, Skill Level, Team, etc.
- **Methods:** Run(), Jump(), KickTheBall(), PassBall(), etc.

Referee:

- **Attributes:** Name, Experience, DecisionMakingAbility, etc.
- **Methods:** ShowCard(), MakeDecision(), etc.

Ball:

- **Attributes:** Size, Color, Velocity, Position, etc.
- **Methods:** Move(), Bounce(), Stop(), etc.

Field:

- **Attributes:** Dimensions, SurfaceType, GoalPosts, etc.
- **Methods:** SetFieldDimensions(), isInsideField(), etc.

Audience:

Classes in a Soccer Game

Player:

- **Attributes:** Name, Number, Location, Skill Level, Team, etc.
- **Methods:** Run(), Jump(), KickTheBall(), PassBall(), etc.

Referee:

- **Attributes:** Name, Experience, DecisionMakingAbility, etc.
- **Methods:** ShowCard(), MakeDecision(), etc.

Ball:

- **Attributes:** Size, Color, Velocity, Position, etc.
- **Methods:** Move(), Bounce(), Stop(), etc.

Field:

- **Attributes:** Dimensions, SurfaceType, GoalPosts, etc.
- **Methods:** SetFieldDimensions(), isInsideField(), etc.

Audience:

- **Attributes:** NumberOfPeople, Cheers, etc.
- **Methods:** Cheer(), etc.

Weather:

Classes in a Soccer Game

Player:

- **Attributes:** Name, Number, Location, Skill Level, Team, etc.
- **Methods:** Run(), Jump(), KickTheBall(), PassBall(), etc.

Referee:

- **Attributes:** Name, Experience, DecisionMakingAbility, etc.
- **Methods:** ShowCard(), MakeDecision(), etc.

Ball:

- **Attributes:** Size, Color, Velocity, Position, etc.
- **Methods:** Move(), Bounce(), Stop(), etc.

Field:

- **Attributes:** Dimensions, SurfaceType, GoalPosts, etc.
- **Methods:** SetFieldDimensions(), isInsideField(), etc.

Audience:

- **Attributes:** NumberOfPeople, Cheers, etc.
- **Methods:** Cheer(), etc.

Weather:

- **Attributes:** Temperature, Humidity, WindSpeed, etc.
- **Methods:** ChangeTemperature(), etc.

OO in Soccer Game: Python illustration

```
class Player:
    def __init__(self, name, number, skill_level, team):
        self.name = name
        self.number = number
        self.skill_level = skill_level
        self.team = team

    def run(self):
        print(f"{self.name} is running")

    def kick_the_ball(self):
        print(f"{self.name} kicks the ball")

    def pass_ball(self):
        print(f"{self.name} passes the ball")

class Ball:
    def __init__(self, size, color):
        self.size = size
        self.color = color
        self.position = (0, 0)

    def move(self, new_position):
        self.position = new_position
        print(f"The ball is now at {self.position}")
```

OO in Soccer Game: Python illustration

```
class Field:
    def __init__(self, dimensions, surface_type):
        self.dimensions = dimensions
        self.surface_type = surface_type

    def is_inside_field(self, position):
        x, y = position
        if 0 <= x <= self.dimensions[0] and 0 <= y <= self.dimensions[1]:
            return True
        return False

class Referee:
    def __init__(self, name, experience):
        self.name = name
        self.experience = experience

    def show_card(self, card_color):
        print(f"{self.name} shows a {card_color} card")

    def make_decision(self, decision):
        print(f"{self.name} makes a decision: {decision}")
```

OO in Soccer Game: Python illustration

```
# Example usage
player1 = Player("John Doe", 10, "High", "Team A")
player2 = Player("Jane Roe", 7, "Medium", "Team B")
ball = Ball(5, "White")
field = Field((100, 60), "Grass")
referee = Referee("Jane Smith", 10)

player1.run()
player1.kick_the_ball()
player2.run()
player2.pass_ball()
ball.move((30, 15))
if field.is_inside_field(ball.position):
    print("The ball is inside the field")
else:
    print("The ball is outside the field")
referee.show_card("yellow")
referee.make_decision("Goal")
```

Object-Oriented Programming: Define Your Own Type

- We have seen different types,
 - `int` for integers
 - `float` for real numbers
- What if, for example, we want to represent rational numbers exactly?
- We can easily define our own type, namely the **`fraction` type**.

Example: A Python class Fraction

```
class Fraction:
    """Represents a fraction.
    Properties: numerator, denominator.
    """

    def __init__(self, numerator=0, denominator=1):
        self.numerator = numerator
        self.denominator = denominator

    def print_fraction(self):
        print(self.numerator, '/', self.denominator)

# Create an instance of the class with default values
# and print it
p = Fraction()
p.print_fraction()

# Create another instance of the class with specified values
# and print it
q = Fraction(1, 3)
q.print_fraction()
```

Example: A Python class Fraction

```
class Fraction:
    """Represents a fraction.
    Properties: numerator, denominator.
    """

    def __init__(self, numerator=0, denominator=1):
        self.numerator = numerator
        self.denominator = denominator

    def print_fraction(self):
        print(self.numerator, '/', self.denominator)

# Create an instance of the class with default values
# and print it
p = Fraction()
p.print_fraction()

# Create another instance of the class with specified values
# and print it
q = Fraction(1, 3)
q.print_fraction()
```

The output is:

0 / 1

1 / 3

- **Class and Objects**

- **Fraction** is a *class* and **p** is an *object*, i.e., an instance of a class.
- A *class* has *properties (attributes)* and *methods*.
- The word *self* refers to the current object (that was called).
- An *attribute* such as `numerator` is individual to the object.
You can change the value via e.g. `p.numerator = 3`.

• Class and Objects

- **Fraction** is a *class* and **p** is an *object*, i.e., an instance of a class.
- A *class* has *properties (attributes)* and *methods*.
- The word **self** refers to the current object (that was called).
- An *attribute* such as numerator is individual to the object.
You can change the value via e.g. `p.numerator = 3`.

• Constructor

- The **constructor** (always defined via `__init__`) is a special method that is called when a new *instance* of the class object is created.
- When called, this method **constructs** a fraction with the given numerator and denominator (the 2nd and 3rd parameters), with a default value of 0/1 if necessary.
- For example: each instance is a different fraction.

Destructor

Python's 'garbage collection' automatically frees up memory for you.

```
class MyClass:
    def __init__(self):
        self.my_list = [1,2,3]

    def __del__(self):
        print("Dying")

inst1 = MyClass()

del inst1 # delete and prints Dying.
```

In the Destructor, you can for example close files / do other clean up.

1 Iterative versus Recursive Algorithms - Memoization

2 Object-Oriented Programming

- Motivation and Terminology
- **Overloading**
- Encapsulation
- Inheritance and Polymorphism

Overloading: str as a standard Method

- Using [overloading](#), we can ensure that several standard operations also work with our new objects, e.g., `+` (`__add__`), `str` (`__str__`), etc.

Overloading: str as a standard Method

- Using [overloading](#), we can ensure that several standard operations also work with our new objects, e.g., + (`__add__`), str (`__str__`), etc.
- The `__str__` method creates a string representation of a fraction (the below is continuation of the previous example with class Fraction):

```
class Fraction:

    def __str__(self):
        return str(self.numerator)+'/'+str(self.denominator)

print(str(p), str(q))
```

- Now, you can apply str not only to an integer or float but also to a fraction.
- The output is
0/1 1/3
- The str method will look for a `__str__` method for a Fraction type.

Overloading: adding fractions

- Fraction addition is done through

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

- Because this is a standard method, we can **overload** the '+' operator and add two fractions (the below is continuation of the previous example with class Fraction):

```
def __add__(self, other):  
    r = Fraction()  
    r.numerator = self.numerator*other.denominator + (  
        self.denominator*other.numerator )  
    r.denominator = self.denominator*other.denominator  
    return r
```

```
p = Fraction(2,3)  
q = Fraction(3,4)  
r = p + q  
print(str(r))
```

- The output is: 17/12

Adding a Fraction and an Integer

- Check the type of the second operand:

```
class Fraction:
    def __add__(self, other):
        r = Fraction()
        if isinstance(other, Fraction):
            r.numerator = self.numerator*other.denominator
            + (self.denominator*other.numerator )
            r.denominator = self.denominator*other.denominator
        elif isinstance(other, int):
            r.numerator = self.numerator
            + self.denominator*other
            r.denominator = self.denominator
        else:
            raise TypeError("+: only fractions, integers")
        return r

p = Fraction(1,3)
n = 5
r = p + n
print(r)
```

- The output is: 16/3

Operator Overloading Possibilities

- Multiplication, division, subtraction (`--sub--`), negation (`--neg--`)
- Conversion to `int` (`--int--`), or to `float`
- Comparison $a < b$ (`--lt--`)
- For more details, refer to the Python documentation and <https://realpython.com/operator-function-overloading/>.

Continued Fraction

- **Continued Fraction**: An iterative process of representing a number as the sum of its integer part and the reciprocal of another number, then writing this other number as the sum of its integer part and another reciprocal, and so on.
- A **regular continued fraction** can be written as (a_i are integers)

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{\ddots}}}}$$

- An example is the **golden ratio**:

$$1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{\ddots}}}} = \frac{1 + \sqrt{5}}{2} \approx 1.6180339.$$

Simplify: No standard Method

- Cancel out the greatest common divisor of the numerator and denominator:

```
def gcd(a,b):  
    # calculates the gcd of a and b, for a,b >= 1  
    if a < b:  
        a,b = b,a  
    while b > 0:  
        a = a % b  
        a,b = b,a  
    return a  
  
class Fraction:  
    def simplify(self):  
        divisor = gcd(self.numerator, self.denominator)  
        self.numerator = self.numerator // divisor  
        self.denominator = self.denominator // divisor  
  
p=Fraction(120,1024)  
p.simplify()  
print(str(p))
```

- The output is: 15/128

Finite Continued Fraction

We can calculate the continued fraction to a finite depth (from low to high):

```
class Fraction:
    def inverse(self):
        self.numerator, self.denominator = self.denominator, self.numerator

a = Fraction(1,1)
p = Fraction(1,1)
print(str(p), '=', p.numerator/p.denominator)
for i in range(15):
    p.inverse()
    p += a
    p.simplify()
    print(str(p), '=', p.numerator/p.denominator)
```

Output Converges to Golden Ratio

- The output is:

$$1/1 = 1.0$$

$$2/1 = 2.0$$

$$3/2 = 1.5$$

$$5/3 = 1.6666666666666667$$

$$8/5 = 1.6$$

$$13/8 = 1.625$$

$$21/13 = 1.6153846153846154$$

$$34/21 = 1.619047619047619$$

$$55/34 = 1.6176470588235294$$

$$89/55 = 1.6181818181818182$$

$$144/89 = 1.6179775280898876$$

$$233/144 = 1.6180555555555556$$

$$377/233 = 1.6180257510729614$$

$$610/377 = 1.6180371352785146$$

$$987/610 = 1.618032786885246$$

$$1597/987 = 1.618034447821682$$

- The golden ratio is:


$$1.618033988749895$$

1 Iterative versus Recursive Algorithms - Memoization


2 Object-Oriented Programming

- Motivation and Terminology
- Overloading
- **Encapsulation**
- Inheritance and Polymorphism

Methods in a class

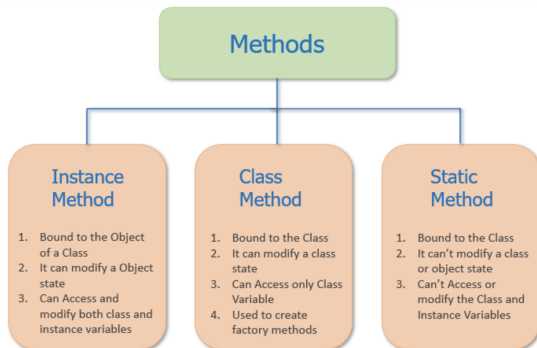
- ① An *instance method*: usually functions defined within the class start with `self` as first input, (e.g. `print_fraction(self)` seen before).
 Such a function can only be called from an object of the class.

Methods in a class

- 1 An *instance method*: usually functions defined within the class start with `self` as first input, (e.g. `print_fraction(self)` seen before).
 Such a function can only be called from an object of the class.
- 2 A *class method* can be called without creating an object. It can access the class variables (but no instance variables).

Methods in a class

- 1 An *instance method*: usually functions defined within the class start with `self` as first input, (e.g. `print_fraction(self)` seen before).
⚠ Such a function can only be called from an object of the class.
- 2 A *class method* can be called without creating an object. It can access the class variables (but no instance variables).
- 3 A *static function* cannot access class variables nor instance variables.



Usage example: class method

```
class Student:
    uni_name = "UU" # class variable

    def __init__(self, name="B"):
        self.name = name # instance variable

    @classmethod
    def change_name(cls, new_name):
        print("Changing name from "+cls.uni_name)
        cls.uni_name = new_name

Student.change_name("UU2") # prints Changing name from UU
s = Student()
s.change_name("UU3") # prints Changing name from UU2
```

Usage example: static method

```
class Student:
    uni_name = "UU" # class variable

    def __init__(self, name="B"):
        self.name = name # instance variable

    @staticmethod
    def print_message(message):
        print(message)

Student.print_message("Hi")      # prints Hi
s = Student()
s.print_message("Hi back")      # prints Hi back
```

- `@classmethod` and `@staticmethod` are python decorators.
- Decorators are a powerful and useful tool in Python since it allows programmers to modify the behaviour of a function or class.
- The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it.
- See this documentation for further details:

[https:](https://python101.pythonlibrary.org/chapter25_decorators.html)

[//python101.pythonlibrary.org/chapter25_decorators.html](https://python101.pythonlibrary.org/chapter25_decorators.html)

Decorator example in Python

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner

@smart_divide
def divide(a, b):
    print(a/b)

divide(2,5)

divide(2,0)
```

Decorator example in Python

```
def smart_divide(func):  
    def inner(a, b):  
        print("I am going to divide", a, "and", b)  
        if b == 0:  
            print("Whoops! cannot divide")  
            return  
  
        return func(a, b)  
    return inner  
  
@smart_divide  
def divide(a, b):  
    print(a/b)  
  
divide(2,5)  
  
divide(2,0)
```

Output

I am going to divide 2 and 5
0.4
I am going to divide 2 and 0
Whoops! cannot divides

Instance, class or static method: When to use which?

- A method defined within a class is automatically an **instance method**, which can access the instance and class variables.

Instance, class or static method: When to use which?

- A method defined within a class is automatically an **instance method**, which can access the instance and class variables.
- **Class** (or **static**) methods do not adjust **instance variables** (**nor class variables**).

Instance, class or static method: When to use which?

- A method defined within a class is automatically an **instance method**, which can access the instance and class variables.
- **Class** (or **static**) methods do not adjust **instance variables** (**nor class variables**).
- General rule: Mark methods as **static** or **class** if they can be.

Instance, class or static method: When to use which?

- A method defined within a class is automatically an **instance method**, which can access the instance and class variables.
- **Class** (or **static**) methods do not adjust **instance variables** (**nor class variables**).
- General rule: Mark methods as **static** or **class** if they can be.
- Functions will often be **instance methods**.

Instance, class or static method: When to use which?

- A method defined within a class is automatically an **instance method**, which can access the instance and class variables.
- **Class** (or **static**) methods do not adjust **instance variables** (**nor class variables**).
- General rule: Mark methods as **static** or **class** if they can be.
- Functions will often be **instance methods**.
- **Class** methods manipulate class data (limit this!)

Instance, class or static method: When to use which?

- A method defined within a class is automatically an **instance method**, which can access the instance and class variables.
- **Class** (or **static**) methods do not adjust **instance variables** (**nor class variables**).
- General rule: Mark methods as **static** or **class** if they can be.
- Functions will often be **instance methods**.
- **Class** methods manipulate class data (limit this!)
- **Static** methods are often utility functions.

Instance, class or static method: When to use which?

- A method defined within a class is automatically an **instance method**, which can access the instance and class variables.
- **Class** (or **static**) methods do not adjust **instance variables** (**nor class variables**).
- General rule: Mark methods as **static** or **class** if they can be.
- Functions will often be **instance methods**.
- **Class** methods manipulate class data (limit this!)
- **Static** methods are often utility functions.

1 Iterative versus Recursive Algorithms - Memoization

2 Object-Oriented Programming

- Motivation and Terminology
- Overloading
- Encapsulation
- Inheritance and Polymorphism

- Sometimes different classes have many similarities.
- In that case, we can let one class **inherit** methods from another class.
- Inheritance: **inheriting the properties of the parent class** (also called base class) into a child class (also called subclass or derived class).
 - Single parent can have multiple children.
 - Single child can inherit from multiple parent classes.
 - Can also have grandparent → parent → child.

Example Python code

```
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle): # Car is child of Vehicle
    def car_info(self, name):
        print("Car name is:", name)

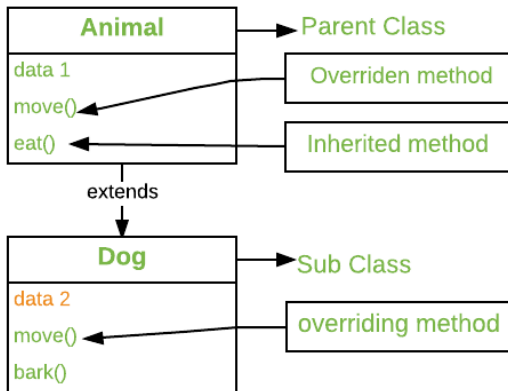
class Truck(Car): # Truck is grandchild of Vehicle
    def truck_info(self, name):
        print("Truck name is:", name)

obj1 = Truck()
obj1.info()
obj1.car_info('BMW')
```


Inheritance

As a child class, you can inherit everything from your parents.

- Child can call the methods and class variables of parent.
- **Method overriding**: you can create a function with the same name and parameters, which Python will use instead for the subclass.



Example: Overriden and inherited methods

```
class Animal:
    def __init__(self, name):
        self.name = name # Public attribute
        self._type = "General Animal" # Protected attribute

    def move(self):
        print(f"{self.name} is moving")

    def eat(self):
        print(f"{self.name} is eating")

    def show_info(self):
        print(f"Name: {self.name}, Type: {self._type}")

# Example usage of the Animal class
animal = Animal("Generic Animal")
animal.move() # Output: Generic Animal is moving
animal.eat() # Output: Generic Animal is eating
animal.show_info() # Output: Name: Generic Animal,
                  #Type: General Animal
```

Example: Overriden and inherited methods (continued)

```
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)    # Call the constructor of
                                   #the base class

        self._type = "Dog"    # Override the protected attribute
        self.breed = breed    # Additional public attribute
                                   #specific to Dog

    # Overriding the move method
    def move(self):
        print(f"{self.name} is running")

    def bark(self):
        print(f"{self.name} is barking")

    def show_info(self):
        print(f"Name: {self.name}, Breed: {self.breed}, Type: {self._type}")

# Example usage of the Dog class
dog = Dog("Buddy", "Golden Retriever")
dog.move()    # Output: Buddy is running
dog.eat()    # Output: Buddy is eating
dog.bark()    # Output: Buddy is barking
dog.show_info()    # Output: Name: Buddy, Breed: Golden Retriever,
                    #Type: Dog
```

Example: use of `super()`

We want a class for fractions of the form $\frac{1}{n}$, without having to redo all the work for writing the methods:

- Option 1:

```
class InverseFraction(Fraction):  
    def __init__(self, denominator=1):  
        self.numerator = 1  
        self.denominator = denominator
```

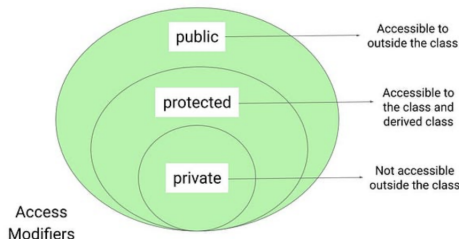
- Option 2 (better): explicitly use the superclass (`super()` used to refer to parent class within child class)

```
class InverseFraction(Fraction):  
    def __init__(self, denominator=1):  
        super().__init__(1, denominator)
```

Public, protected, private

We can choose from where instance variables/methods can be accessed.

- **Public**: accessible from anywhere, standard.
- **Protected**: accessible from within the class and its sub-classes.
Need to add one underscore before name.
- **Private**: accessible from within the class only.
Need to add two underscores at front, e.g. `__salary`.



Example: private and protected variables

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary # Private variable
        self._performance_score = 0 # Protected variable

    def get_salary(self):
        # Controlled access to private variable
        return self.__salary

    def set_performance_score(self, score):
        # Controlled access to protected variable
        if 0 <= score <= 100:
            self._performance_score = score
        else:
            raise ValueError("Performance score must...\
                               ... be between 0 and 100.")

    def display_details(self):
        print(f"Name: {self.name}, Salary: ${self.get_salary()},...\
              Performance: {self._performance_score}")
```

Example: private and protected variables

```
class Manager(Employee):
    def __init__(self, name, salary, bonus):
        super().__init__(name, salary)
        self._bonus = bonus # Protected variable for subclass use

    def calculate_total_compensation(self):
        # Can access protected variables (_performance_score and _bonus)
        bonus_multiplier = 1 + (self._performance_score / 100)
        return self.get_salary() + (self._bonus * bonus_multiplier)

    def display_details(self):
        # Extend base class functionality
        total_compensation = self.calculate_total_compensation()
        print(f"Name: {self.name}, Total Compensation: ${total_compensation},...\n"
              f"Performance: {self._performance_score}")
```

Example: private and protected variables

```
# Usage
# Base Employee
employee = Employee("Alice", 50000)
employee.set_performance_score(85)
employee.display_details()

# Manager with extended functionality
manager = Manager("Bob", 80000, 10000)
manager.set_performance_score(90)
manager.display_details()

# Access attempts
print("\nDirect Access Attempts:")
try:
    print(employee.__salary) # Raises AttributeError
except AttributeError:
    print("Cannot access private variable '__salary' directly.")

print(manager._performance_score) # Allowed but discouraged
```


More about Classes and Object-Oriented Programming

- <https://docs.python.org/3.12/tutorial/classes.html>