

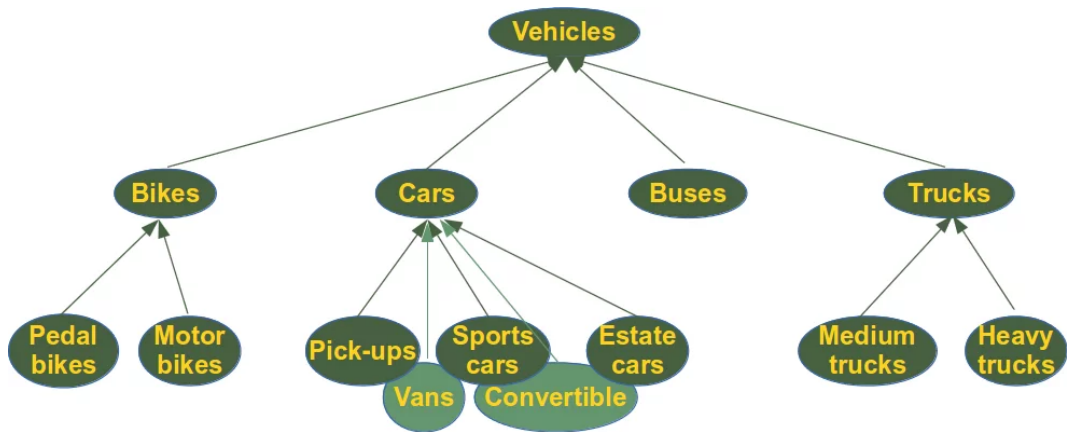
# Data Science and Advanced Programming — Lecture 6c

## Object Oriented Programming

Simon Scheidegger  
Department of Economics, University of Lausanne, Switzerland

October 20th, 2025 | 10:15 - 14:00 | Anthropole 2106

# 1. Python Classes and Inheritance



# Implementing the Class vs using the Class

Write code from two different perspectives:

- ▶ **implementing** a new object type with a class
  - ▶ **define** the class
  - ▶ define **data attributes** (WHAT IS the object)
  - ▶ **define methods** (HOW TO use the object)
- ▶ **using** the new object type in Code
  - ▶ create **instances** of the object type
  - ▶ do **operations** with them

# Class definition of an object type versus instance of a class

- ▶ Class name is the type:  
`class Coordinate(object)`
- ▶ Class is defined generically
- ▶ use `self` to refer to some instance while defining the class  
`(self.x - self.y)**2`
- ▶ `self` is a parameter to methods in the class definition.
- ▶ class defines **data and methods common across all instances.**
- ▶ **instance** is **one specific object**  
`coord = Coordinate(1,2)`
- ▶ Data attribute values may vary between instances  
`c1 = Coordinate(1,2)`  
`c2 = Coordinate(3,4)`
  - ▶ `c1` and `c2` can have different data attribute values `c1.x` and `c2.x` **because they are different objects.**
- ▶ Instance has the **structure of the class.**



# Why should we use OOP and classes of objects?

- ▶ mimic real life
- ▶ **group** different objects part of the same type (e.g. car: age, name).

Object "car".  
Age: 70y  
Name  
associated  
with it:  
"Oldtimer"



10y old plane

90y old plane



2y old car

# Why should we use OOP and classes of objects?

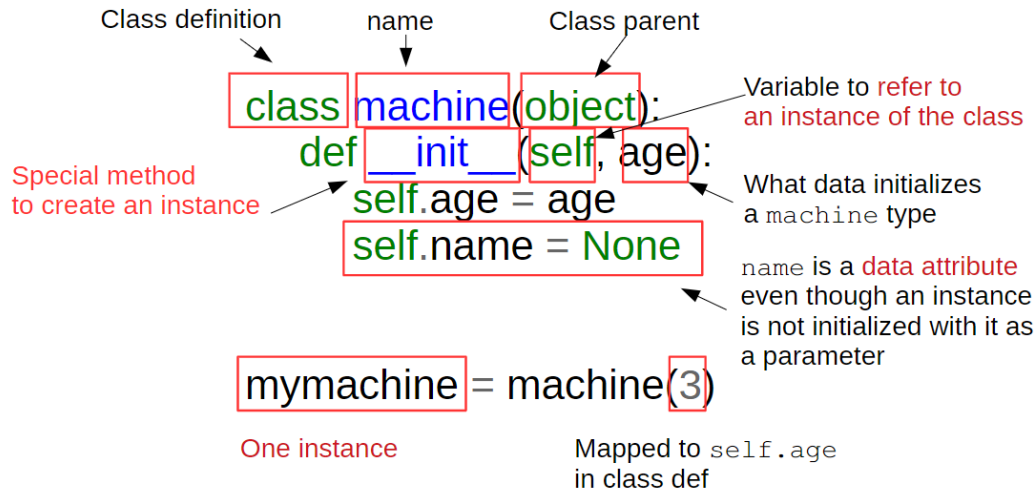
- ▶ mimic real life
- ▶ **group** different objects part of the same type (e.g. car: age, name).



# Recall: Groups of Objects have attributes

- ▶ **data attributes**
  - ▶ how can you represent your object with data?
  - ▶ **what it is**
  - ▶ for a *coordinate*:  $x$  and  $y$  values
  - ▶ for a *car*: e.g. age, name
- ▶ **procedural attributes** (behavior/operations/**methods**)
  - ▶ how can someone interact with the object?
  - ▶ **what it does/what can your object do?**
  - ▶ for a *coordinate*: find distance between two
  - ▶ for a *car*: make a sound (horn)

# Recall: How to define a class



# Getter” and “Setter” Methods

demo/example4.py

```
class machine(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "Machine:" + str(self.name) + ":" + str(self.age)
```

Getter

Setter

- ▶ **getters** and **setters** should be used outside of class to access data attributes.
- ▶ Implementing getters and setters help to prevent from introducing bugs.

## Recall: An instance — not notation

- ▶ Instantiation creates an instance of an object

```
a = machine(3)
```

- ▶ Dot notation used to access attributes (data and methods) though it is better to use getters and setters to access data attributes
- ▶ `a.age`
  - ▶ → access data attribute
  - ▶ → allowed, but not recommended
- ▶ `a.get_age()`
  - ▶ → access method
  - ▶ → **best to use getters and setters**

# Information hiding

- ▶ Author of class definition may **change data attribute** variable names

Replaced age data  
attribute by years

```
class machine(object):  
    def __init__(self, age):  
        self.years = age  
    def get_age(self):  
        return self.years
```

- ▶ If you are accessing data attributes outside the class and class definition changes, **may get errors**
- ▶ outside of class, **use getters and setters** instead use **a.get\_age()** NOT **a.age**
  - ▶ good style
  - ▶ easy to maintain code
  - ▶ prevents bugs

# Python: Not so good at information hiding

- ▶ Allows you to **access data from outside class** definition  
`print(a.age)`
- ▶ Allows you to **write to data from outside class** definition  
`(int ↔ str) a.age = 'infinite'`
- ▶ Allows you to **create data attributes** for an instance from outside class definition  
(e.g., `size`)  
`a.size = "tiny"`

→ It's **not good style** to do any of these!



# Default arguments

- ▶ Default arguments for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):  
    self.name = newname
```

- ▶ Default argument used here

```
a = machine(3)  
a.set_name()  
print(a.get_name())
```

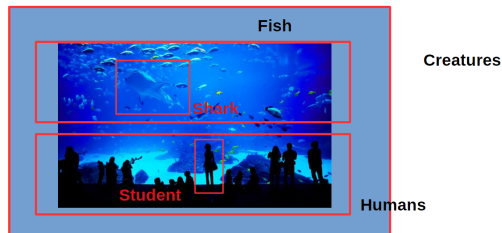
- ▶ argument passed in is used here

```
a = machine(3)  
a.set_name("airbus")  
print(a.get_name())
```

- ▶ Prints " "

- ▶ Prints "airbus"

# Hierarchies

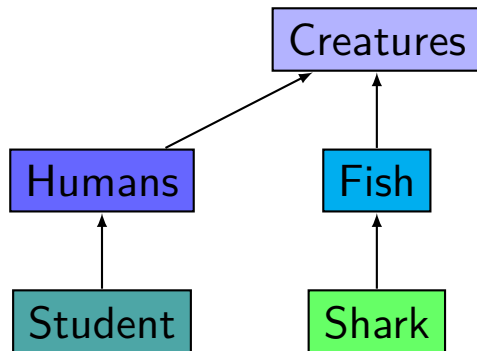


Consider everything on this picture as an Object!

- ▶ Every Creature has an age
- ▶ Fish and Humans are Creatures. Humans have different (data) attributes than Fish.
- ▶ Shark builds on Object Fish.
- ▶ Student builds on Object Human. (Student is Creature, is human, has an age, and Major field of study...)
- ▶ Idea of HIERARCHY.
- ▶ **Add functionality to each of those “subgroups”**

# Hierarchies

- ▶ **parent class** (**superclass**)
- ▶ **child class** (**subclass**)
  - ▶ **inherits** all data and methods/behaviors of parent class.
  - ▶ **add more info.**
  - ▶ **add more behavior.** (e.g. student study, while sharks do not)
  - ▶ **override behavior.** (e.g. person can speak, but say only “hello”, while a student can say “hi dude” → override “speak” function)



# Example of inheritance: Parent class

demo/example5.py

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:" + str(self.name) + ":" + str(self.age)
```

- ▶ everything is an **object**
- ▶ class object implements basic operations in Python, such as binding variables, etc.

# Inheritance: Subclass "Cat"

demo/example5.py

Inherits all attributes of Animal:

```
__init__()  
age.name  
get_age(), get_name()  
set_age(), set_name()  
__str__()
```

Add new functionality  
by introducing the method  
"speak"

```
class Cat(Animal):
```

```
    def speak(self):
```

```
        print("meow")
```

```
    def __str__(self):
```

```
        return "cat:"+str(self.name)+":"+str(self.age)
```

Overrides `__str__()`

- ▶ add new method/functionality with `speak()`.
- ▶ instance of type cat can be called with new methods.
- ▶ instance of type Animal throws error if called with Animal's new method.
- ▶ `__init__` is not missing, uses the Animal version.

# Which method to use

- ▶ Subclass can have **methods with same name** as superclass.
- ▶ For an instance of a class, look for a method name in **current class definition**.
- ▶ If not found, look for method name up the hierarchy (**in parent, then grandparent, and so on**).
- ▶ **Use first method up the hierarchy** that you found with that method name.

# Example of inheritance: Parent class

demo/example5.py

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:" + str(self.name) + ":" + str(self.age)
```

# Example of Inheritance — Person

demo/example6.py

```
class Person(Animal):  
    def __init__(self, name, age):  
        Animal.__init__(self, age)  
        self.set_name(name)  
        self.friends = []  
    def get_friends(self):  
        return self.friends  
    def add_friend(self, fname):  
        if fname not in self.friends:  
            self.friends.append(fname)  
    def speak(self):  
        print("hello")  
    def age_diff(self, other):  
        diff = self.age - other.age  
        print(abs(diff), "year difference")  
    def __str__(self):  
        return "person:" + str(self.name) + ":" + str(self.age)
```

parent class is Animal

call Animal constructor  
call Animal's method  
add a new data attribute

new methods

override Animal's  
\_\_str\_\_ method



# Example of Inheritance — Student (Subclass of Person, i.e., sub-subclass of Animal)

demo/example7.py

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in methods  
from random class  
inherits Person and  
Animal attributes  
adds new data

- I looked up how to use the  
random class in the python docs  
- random() method gives back  
float in [0, 1)

# Class Variables

demo/example8.py

- **class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal): ← Parent class
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Class Variable → tag = 1

Instance variable → self.rid = Rabbit.tag

Access class variable

Incrementing class variable changes it for all instances that may reference it.

- Tag used to give **unique ID** to each new rabbit instance

# Rabbit "Getter" methods

demo/example8.py

```
class Rabbit(Animal):  
    tag = 1  
    def __init__(self, age, parent1=None, parent2=None):  
        Animal.__init__(self, age)  
        self.parent1 = parent1  
        self.parent2 = parent2  
        self.rid = Rabbit.tag  
        Rabbit.tag += 1  
  
    def get_rid(self):  
        return str(self.rid).zfill(3)  
  
    def get_parent1(self):  
        return self.parent1  
  
    def get_parent2(self):  
        return self.parent2
```

method on a string to pad  
the beginning with zeros  
for example, 001 not 1

- getter methods specific  
for a Rabbit class  
- there are also getters  
get\_name and get\_age  
inherited from Animal

# Working with your own types

- ▶ define + operator between two Rabbit instances
  - ▶ define what something like this does: `r4= r1+r2` where `r1` and `r2` are Rabbit instances.
  - ▶ `r4` is a new Rabbit instance with age 0
  - ▶ `r4` has `self` as one parent and `other` as the other parent
  - ▶ in `__init__`, `parent1` and `parent2` are of type Rabbit

```
def __add__(self, other):  
    # returning object of same type as this class  
    return Rabbit(0, self, other)
```

recall Rabbit's `__init__(self, age, parent1=None, parent2=None)`



The diagram shows a red box around the arguments `0, self, other` in the `return` statement. Three red arrows point from these arguments down to the parameters `age`, `parent1`, and `parent2` in the `__init__` method signature mentioned in the text below.

# Special Method to Compare two Rabbits

- ▶ Decide that two rabbits are equal if they have the **same two parents**.

```
def __eq__(self, other):  
    parents_same = self.parent1.rid == other.parent1.rid \  
                  and self.parent2.rid == other.parent2.rid  
    parents_opposite = self.parent2.rid == other.parent1.rid \  
                      and self.parent1.rid == other.parent2.rid  
    return parents_same or parents_opposite
```

*booleans*

- ▶ **Compare IDs of parents since IDs are unique** (due to class var)
- ▶ note you can't compare objects directly
  - ▶ for ex. with `self.parent1 == other.parent1`
  - ▶ this calls the `__eq__` method over and over until call it on `None` and gives an `AttributeError` when it tries to do `None.parent1`

# Object-oriented Programming

- ▶ Create your own collections of data.
- ▶ Organize information.
- ▶ Division of work.
- ▶ Access information in a consistent manner.
- ▶ Add layers of complexity.
- ▶ Like functions, classes are a mechanism for decomposition and abstraction in programming.

## 2. Program Efficiency

- ▶ Measuring orders of growth of algorithms.
- ▶ Big “O” notation.
- ▶ Complexity classes.

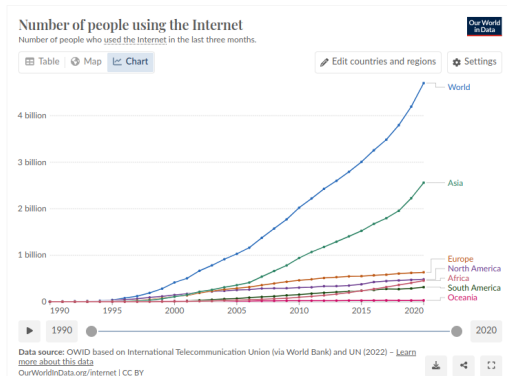
“We are drowning in information and starving for knowledge.”

— John Naisbitt

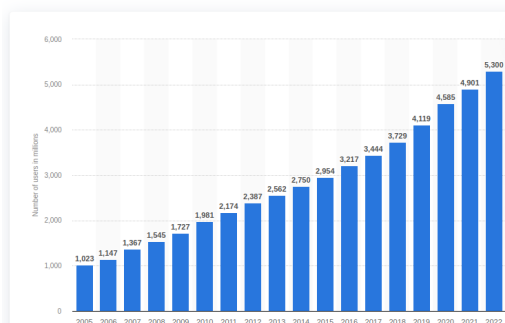


# Big Data and its availability

<https://ourworldindata.org/internet>



**Number of internet users worldwide from 2005 to 2022**  
(in millions)



# Big Data and its availability

<http://www.live-counter.com/how-big-is-the-internet>

Size of the internet as we speak: TBD Petabytes

- ▶ 1 Gigabyte  $\sim$  1000 MB
- ▶ 1 Terabyte  $\sim$  1000 GB
- ▶ 1 Petabyte  $\sim$  1000 TB
- ▶ 1 Exabyte  $\sim$  1000 PB
- ▶ 1 Zettabyte  $\sim$  1000 EB

**1 Gigabyte:** If an author writes a book of **about 190 pages**, more specifically, of 383,561 characters (with spaces and punctuation included) **every week** for **50 years** — this would be a billion letters or bytes.

**1 Exabyte:** 212 million DVDs weighing 3,404 tons.

**1 Zettabyte:** 1,000,000,000,000,000,000 bytes or characters.

This, printed on graph paper (with one letter in each  $\text{mm}^2$  square) would be a paper measuring a billion km. The entire surface of the Earth ( $510 \text{ million km}^2$ ) would be covered by a layer of paper almost twice.

# Other sources of Big Data

## ► Scientific experiments

- CERN (e.g., LHC) generates  $\sim$  **25 petabytes** per year (2012).
- LIGO generates  $\sim$  **1 Petabyte** per year

## ► Numerical computations

► ...



<https://www.olcf.ornl.gov/summit/>



<https://home.cern/>



<https://www.ligo.caltech.edu/>

# Efficiency of Programs

- ▶ **Computers getting faster and faster** — so maybe efficient programs don't matter?
  - ▶ But data sets can be very large (growing exponentially, and **faster than computer power**)
- ▶ Thus, simple solutions may simply not scale with size in acceptable manner.
- ▶ **How can we decide which option for program is most efficient?**
  - ▶ → **Algorithmic Complexity**, **Parallel programming** (later in this course)
- ▶ Separate time and space efficiency of a program
  - ▶ → trade-off between them:
  - ▶ → can sometimes precompute results are stored;
  - ▶ → will focus on time efficiency

# Understanding Efficiency

- ▶ Challenges in understanding efficiency of solution to a computational problem:
  - ▶ A program can be **implemented in many different ways**.
  - ▶ You can **solve a problem** using only a handful of **different algorithms**.
  - ▶ would like to **separate choices of implementation** from choices of more abstract **algorithm**.

# How to evaluate Efficiency

- ▶ Measure with a time
- ▶ count the operations
- ▶ Abstract notation of order of growth  $O()$  - “big O”

→ we argue that this is the most appropriate way of assessing the impact of choices of algorithm in solving a problem, and in measuring the inherent difficulty in solving a problem.

→ Are there fundamental limitations when trying to solve a problem computationally?

# Timing a Program

demo/example9.py

- ▶ recall that importing means to bring this class into your own file
- ▶ use the `time` module

```
import time
```

```
def c_to_f(c):  
    return c*9/5 + 32
```

- **start** clock → `t0 = time.clock()`
- **call** function → `c_to_f(100000)`
- **stop** clock → `t1 = time.clock() - t0`  
`Print("t =", t, ":", t1, "s,")`

# Timing a program is inconsistent

- ▶ GOAL: to evaluate different algorithms
  - ▶ Running time **varies between algorithms.** ✓
  - ▶ Running time **varies between implementations.** ✗
  - ▶ Running time **varies between computers.** ✗
  - ▶ Running time is **not predictable** based on small inputs. ✗

→ Time varies for different inputs but cannot really express a relationship between inputs and time. ✗



# Counting operations

- ▶ Assume these steps take **constant time**:
  - ▶ Mathematical operations.
  - ▶ Comparisons
  - ▶ Assignments
  - ▶ accessing objects in memory
- ▶ Then count the number of operations executed as function of size of input.

```
def c_to_f(c):  
    return c*9.0/5 + 32
```

3 ops

```
def mysum(x):  
    total = 0  
    for i in range(x+1):  
        total += i  
    return total
```

1 op  
loop x times  
2 ops  
1 op

mysum  $\rightarrow 1+3x$  ops

# Counting operations is better, but still not optimal

- ▶ GOAL: to evaluate different algorithms
  - ▶ count **depends on algorithm.** ✓
  - ▶ count **depends on implementations.** ✗ (e.g., “for” vs. “while”)
  - ▶ count **independent of computers.** ✓
  - ▶ no clear definition of **which operations to count.** ✗
  - ▶ count varies for different inputs and can come up with a relationship between inputs and the count. ✓

# Need a better way

- ▶ Timing and counting **evaluate implementations.**
- ▶ Timing **evaluates machines.**
- ▶ → Want to **evaluate algorithm.**
- ▶ → Want to **evaluate scalability.**
- ▶ → Want to **evaluate in terms of input size.**

# Need a better way (II)

- ▶ Going to focus on idea of **counting operations** in an algorithm, but **not worry about small variations in implementation** (e.g., whether we take or 4 primitive operations to execute the steps of a loop).
- ▶ Going to focus on **how an algorithm performs when size of problem gets arbitrarily large.**
- ▶ Want to **relate time needed** to complete a computation, measured this way, **against the size of the input** to the problem.
- ▶ Need to decide **what to measure**, given that actual number of steps may depend on specifics of trial.

# Need to choose which input to use to evaluate a function

- ▶ Want to express **efficiency in terms of size of input**, so need to decide what your input is.
- ▶ Could be an **integer** — `mysum(x)`
- ▶ Could be **length of list** — `list_sum(L)`
- ▶ **You decide** when multiple parameters to a function — `search_for_elemt(L, e)`

# Different inputs change how the programs run

demo/example10.py

- ▶ A function that searches for an element in a list

```
def search_for_elmt(L, e):  
    for i in L:  
        if i == e:  
            return True  
    return False
```

- ▶ When e is **first element** in the list → **BEST CASE**.
- ▶ When e is **not in list** → **WORST CASE**.
- ▶ When we have to look **through about half of the elements** in list → **AVERAGE CASE**.

Want to measure this behavior in a general way.

# Best, average, and worst cases

Suppose you are given a list  $L$  of some length  $\text{len}(L)$

- ▶ **Best case**: minimum running time over all possible inputs of a given size,  $\text{len}(L)$ 
  - ▶ Constant for `search_for_elt`.
  - ▶ First element in any list.
  - ▶ **Average case**: average running time over all possible inputs of a given size,  $\text{len}(L)$
  - ▶ Practical measure
- ▶ **Worst case**: maximum running time over all possible inputs of a given size,  $\text{len}(L)$  ← **We usually focus on this case**
  - ▶ linear in length of list for `search_for_elt`
  - ▶ must search entire list and not find it.

# Orders of growth

## Goals:

- ▶ We want to evaluate program's efficiency when **input is very big**.
- ▶ We want to express the **growth of program's runtime** as input size grows.
- ▶ We want to put an **upper bound** on growth-as tight as possible.
- ▶ We do not need to be precise: "**order of**" not "**exact**" growth.
- ▶ We will look at **largest factors** in runtime (which section of the program will take the longest to run?)
- ▶ **Generally we want tight upper bound on growth, as function of size of input, in worst case.**



# Measuring the order of growth: The Big-O-notation

- ▶ *Big O* notation measures an **upper bound on the asymptotic growth**, often called order of growth.
- ▶ *Big O* or  $O()$  is used to describe **worst case**
  - ▶ worst case that occurs; is the bottleneck when a program runs.
  - ▶ express rate of growth of program relative to the input size.
  - ▶ Evaluates algorithm, NOT machine or implementation.

# Exact steps vs $O()$

```
def fact_iter(n):  
    """assumes n an int >= 0"""  
    answer = 1  
    while n > 1:  
        answer *= n  
        n -= 1  
    return answer
```

answer = answer \* n  
temp = n - 1  
n = temp

- ▶ Computes factorial
- ▶ Number of steps:  $1 + 5n + 1$
- ▶ worst case **asymptotic complexity:**  $O(n)$
- ▶ Ignore additive constants
- ▶ Ignore multiplicative constants

# What does $O(N)$ measure

- ▶ Interested in describing how the amount of time needed grows as size of (input to) problem grows.
- ▶ Thus, given an expression for the number of operations needed to compute an algorithm, want to know asymptotic behavior as size of problem gets large.
- ▶ We will focus on term that grows most rapidly in a sum of terms.
- ▶ Will ignore multiplicative constants, since want to know how rapidly time required increases as increase size of input.

# Examples

- ▶ Drop constants and multiplicative factors
- ▶ Focus on dominant terms

$$O(n^2) : n^2 + 2n + 2$$

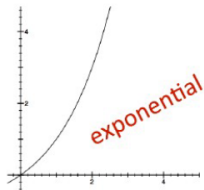
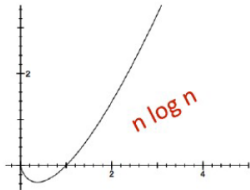
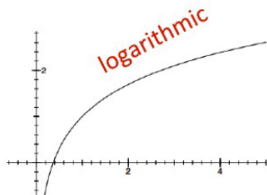
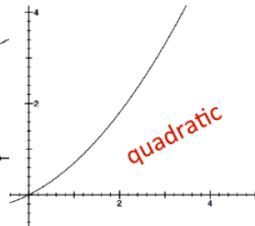
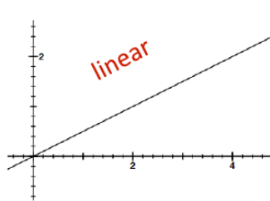
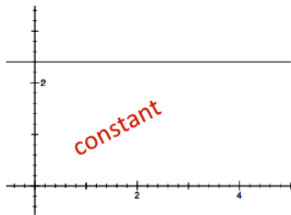
$$O(n^2) : n^2 + 100000n + 3^{1000}$$

$$O(n) : \log(n) + n + 4$$

$$O(n \log n) : 0.0001 * n * \log(n) + 300n$$

$$O(3^n) : 2n^{30} + 3^n$$

# Some examples for growth order



# Analyzing Programs and their algorithmic complexity

- ▶ **Combine** complexity classes
- ▶ Analyze statements inside functions
- ▶ Apply some rules, focus on dominant term
- ▶ **Law of addition for  $O()$**  :
- ▶ Used with **sequential** statements
- ▶  $O(f(n)) + O(g(n))$  is  $O(f(n) + g(n))$
- ▶ for example,

```
for i in range(n):  
    print('a')  
for j in range(n*n):  
    print('b')
```

$O(n)$

$O(n^2)$

$O(n) + O(n^2)$

- ▶ is  $O(n) + O(n^2) = O(n + n^2) = O(n^2)$  because of dominant term.

# Analyzing Programs and their algorithmic complexity (II)

demo/example12.py

- ▶ Combine complexity classes
  - ▶ Analyze statements inside functions
  - ▶ Apply some rules, focus on dominant term
- ▶ **Law of Multiplication for  $O()$  :**
  - ▶ Used with **nested** statements/loops
  - ▶  $O(f(n)) * O(g(n))$  is  $O(f(n) * g(n))$
  - ▶ for example,

```
for i in range(n):  
    for j in range(n):  
        print('a')
```

}  $O(n)$

}  $n$  loops, each  $O(n) \rightarrow O(n) * O(n)$

- ▶ -This is  $O(n) * O(n) = O(n * n) = O(n^2)$  because the outer loop goes  $n$  times and the inner loop goes  $n$  times for every outer loop iter.

# Typical complexity classes

- ▶  $O(1)$  denotes constant running time.
- ▶  $O(\log n)$  denotes logarithmic running time.
- ▶  $O(n)$  denotes linear running time.
- ▶  $O(n \log n)$  denotes log-linear running time.
- ▶  $O(n^c)$  denotes polynomial running time ( $c$  is a constant).
- ▶  $O(c^n)$  denotes exponential running time ( $c$  is a constant being raised to a power based on size of input).



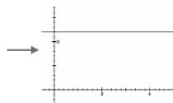
# Complexity classes — ordered

*c is a  
constant*

$O(1)$

:

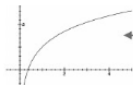
constant



$O(\log n)$

:

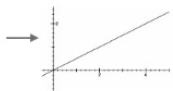
← logarithmic



$O(n)$

:

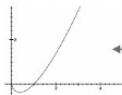
linear



$O(n \log n)$

:

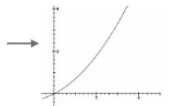
← loglinear



$O(n^c)$

:

polynomial




$O(c^n)$

:

← exponential



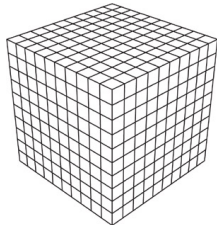
# Complexity Growth



CLASS	n=10	= 100	= 1000	= 1000000
$O(1)$	1	1	1	1
$O(\log n)$	1	2	3	6
$O(n)$	10	100	1000	1000000
$O(n \log n)$	10	200	3000	6000000
$O(n^2)$	100	10000	1000000	1000000000000
$O(2^n)$	1024	12676506 00228229 40149670 3205376	1071508607186267320948425049060 0018105614048117055336074437503 8837035105112493612249319837881 5695858127594672917553146825187 1452856923140435984577574698574 8039345677748242309854210746050 6237114187795418215304647498358 1941267398767559165543946077062 9145711964776865421676604298316 52624386837205668069376	Good luck!!

# Exponential Complexity Example – The Curse of Dimensionality

Number of parameters (the dimension)	Number of model runs (at 10 points per dimension)	Time for parameter study (at 1 second per run)
1	10	10 sec
2	100	~ 1.6 min
3	1,000	~ 16 min
4	10,000	~ 2.7 hours
5	100,000	~ 1.1 days
6	1,000,000	~ 1.6 weeks
...	...	...
20	1e20	3 trillion years (240x age of the universe)



# Linear complexity — e.g. linear search

demo/example13.py

- ▶ Simple iterative loop algorithms are typically linear in complexity.
- ▶ Example: Linear search on an unsorted list.

```
def linear_search(L, e):  
    found = False  
    for i in range(len(L)):  
        if e == L[i]:  
            found = True  
    return found
```

speed up a little by  
returning True here,  
but speed up doesn't  
impact worst case

- ▶ Must look through all elements to decide it's not there
- ▶  $O(\text{len}(L))$  for the loop \*  $O(1)$  to test if  $e == L[i]$  (assumes we can retrieve element of list in constant time)
- ▶  $O(1 + 4n + 1) = O(4n + 2) = O(n)$
- ▶ Overall complexity is  $O(n)$  - where  $n$  is  $\text{len}(L)$

# Linear search on sorted List

demo/example14.py

```
def search(L, e):  
    for i in range(len(L)):  
        if L[i] == e:  
            return True  
        if L[i] > e:  
            return False  
    return False
```

- ▶ Must only look until reach a number greater than  $e$
- ▶  $O(\text{len}(L))$  for the loop\*  $O(1)$  to test if  $e == \text{Len}[i]$
- ▶ Overall complexity is  $O(n)$  — where  $n$  is  $\text{len}(L)$  (worst case we need to look at the entire list).
- ▶ NOTE: order of growth is same, though runtime may differ for two search methods.

# Linear Complexity

demo/example15.py

- ▶ Searching a list in sequence to see if an element is present.
- ▶ Add characters of a string, assumed to be composed of decimal digits.

```
def addDigits(s):  
    val = 0  
    for c in s:  
        val += int(c)  
    return val
```

$O(\text{len}(s))$

# Linear Complexity (II)

- Complexity often depends on number of iterations.

```
def fact_iter(n):  
    prod = 1  
    for i in range(1, n+1):  
        prod*=i  
    return prod
```

- **Number of times around loop is  $n$** 
  - Number of operations inside loop is a constant (in this case, 3-set i, multiply, set prod).
  - $O(1 + 3n + 1) = O(3n + 2) = O(n)$ .
  - Overall just  $O(n)$ .

# Nested Loops — quadratic complexity

- ▶ Simple loops are linear in complexity.
- ▶ What about loops that have loops within them?
- ▶ Example: determine if one list is subset of second, i.e., every element of first, appears in second (assume no duplicates).

```
def isSubset(L1, L2):  
    for e1 in L1:  
        matched = False  
        for e2 in L2:  
            if e1 == e2:  
                matched = True  
                break  
        if not matched:  
            return False  
    return True
```



# Quadratic Complexity (II)

```
def isSubset(L1, L2):  
    for e1 in L1:  
        matched = False  
        for e2 in L2:  
            if e1 == e2:  
                matched = True  
                break  
        if not matched:  
            return False  
    return True
```

- ▶ outer loop executed  $\text{len}(L1)$  times
- ▶ each iteration will execute inner loop up to  $\text{len}(L2)$  times, with constant number of operations
- ▶  $O(\text{len}(L1) * \text{len}(L2))$
- ▶ worst case when  $L1$  and  $L2$  same length, none of elements of  $L1$  in  $L2$
- ▶  $O(\text{len}(L1)^2)$

# Quadratic Complexity (III)

demo/example17.py

**Find intersection of two lists**, return a list with each element appearing only once!

```
def intersect(L1, L2):  
    tmp = []  
    for e1 in L1:  
        for e2 in L2:  
            if e1 == e2:  
                tmp.append(e1)  
    res = []  
    for e in tmp:  
        if not(e in res):  
            res.append(e)  
    return res
```

# Quadratic Complexity (IV)

demo/example17.py

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

- ▶ first nested loop takes  $\text{len}(L1) * \text{len}(L2)$  steps
- ▶ second loop takes at most  $\text{len}(L1)$  steps
- ▶ determining if element in list might take  $\text{len}(L1)$  steps
- ▶ if we assume lists are of roughly same length, then  $O(\text{len}(L1)^2)$

# $O()$ for nested loops

```
def g(n):  
    """ assume n >= 0 """  
    x = 0  
    for i in range(n):  
        for j in range(n):  
            x+=1  
    return x
```

- ▶ Computes  $n^2$  very inefficiently.
- ▶ When dealing with nested loops, look at the ranges.
- ▶ Nested loops, each iterating  $n$  times.
- ▶  $O(n^2)$

# Logarithmic Complexity — Bisection example

- ▶ **Complexity grows as log of size of one of its inputs**
  - ▶ example:
    - ▶ Bisection search
    - ▶ binary search of a list
- ▶ **Bisection search**: suppose we want to know if a particular element is present in a list
- ▶ **Saw that we could just “walk down” the list, checking each element**
  - ▶ Complexity was linear in length of the list
  - ▶ Suppose we know that the **list is ordered from smallest to largest**
  - ▶ Saw that sequential search was still linear in complexity
  - ▶ Can we do better?

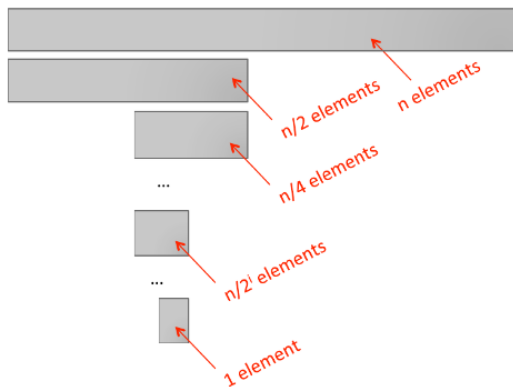
# Yes we can — Bisection search

1. pick an index,  $i$ , that divides list in half
2. ask if  $L[i] == e$
3. if not, ask if  $L[i]$  is larger or smaller than  $e$
4. Depending on answer, search left or right half of  $L$  for  $e$



- ▶ A new version of a **divide-and-conquer** algorithm
  - ▶ Break into smaller version of problem (smaller list), plus some simple operations
  - ▶ Answer to smaller version is answer to original problem.

# Complexity Analysis of Bisection



- finish looking through list when

$$1 = n/2^i$$
$$\text{so } i = \log n$$

- complexity of recursion is  $O(\log n)$ —  
where  $n$  is  $\text{len}(L)$

# Bisection Code

demo/example17.py

```
def bisection_search1(L, e):
```

```
    if L == []:
```

```
        return False
```

```
    elif len(L) == 1:
```

```
        return L[0] == e
```

```
    else:
```

```
        half = len(L)//2
```

```
        if L[half] > e:
```

```
            return bisection_search1(L[:half], e)
```

```
        else:
```

```
            return bisection_search1(L[half:], e)
```

constant  
 $O(1)$

constant  
 $O(1)$

constant  
 $O(1)$

NOT constant,  
copies list

NOT constant

NOT constant

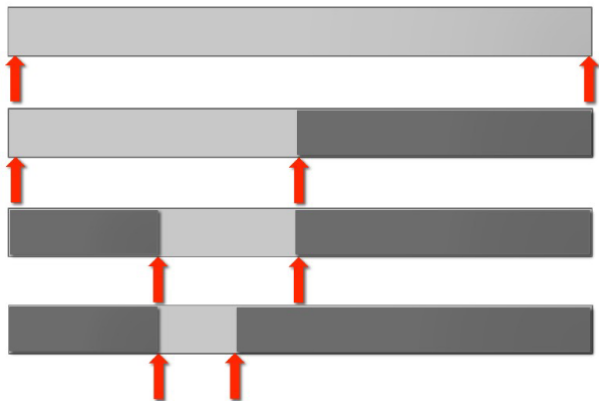


# Complexity of first Bisection Method

Implementation 1 — `bisect_search1` ([demo/example17.py](#))

- ▶  $O(\log n)$  bisection search calls
  - ▶ On each recursive call, size of range to be searched is cut in half
  - ▶ If original range is of size  $n$ , in worst case down to range of size 1 when  $n / (2^k) = 1$ ; or when  $k = \log n$
- ▶  $O(n)$  for each bisection search call to copy list
  - ▶ This is the cost to set up each call, so do this for each level of recursion
- ▶  $O(\log n) * O(n) \rightarrow O(n \log n)$
- ▶ If we are really careful, note that length of list to be copied is also halved on each recursive call.
  - ▶ Turns out that total cost to copy is  $O(n)$  and this dominates the  $\log n$  cost due to the recursive calls.

# An alternative Bisection algorithm



- ▶ still reduce size of problem by factor of two on each step but just keep track of low and high portion of list to be searched
- ▶ avoid copying the list
- ▶ complexity of recursion is again  $O(\log n)$  - where  $n$  is  $\text{len}(L)$

# Bisection (II) Code

demo/example18.py

```
def bisect_search2(L, e):  
    def bisect_search_helper(L, e, low, high):  
        if high == low:  
            return L[low] == e  
        mid = (low + high)//2  
        if L[mid] == e:  
            return True  
        elif L[mid] > e:  
            if low == mid: #nothing left to search  
                return False  
            else:  
                return bisect_search_helper(L, e, low, mid - 1)  
        else:  
            return bisect_search_helper(L, e, mid + 1, high)  
    if len(L) == 0:  
        return False  
    else:  
        return bisect_search_helper(L, e, 0, len(L) - 1)
```

constant other  
than recursive call

constant other  
than recursive call

# Algorithmic Complexity — Bisect II

demo/example18.py

- ▶ Implementation 2 — `bisect_search2` and its helper
  - ▶  $O(\log n)$  bisection search calls
  - ▶ On each recursive call, size of range to be searched is cut in half if original range is of size  $n$ , in worst case down to range of size 1 when  $n / (2^k) = 1$ ; or when  $k = \log n$
- ▶ Pass list and indices as parameters
  - ▶ list never copied, just re-passed as a pointer
  - ▶ Thus  $O(1)$  work on each recursive call
  - ▶  $O(\log n) * O(1) \rightarrow O(\log n)$

# Exponential Complexity

- ▶ Recursive functions where more than one recursive call for each size of problem
  - ▶ Towers of Hanoi
- ▶ Many important problems are inherently exponential
  - ▶ Unfortunately, as cost can be high will lead us to consider approximate solutions as may provide reasonable answer more quickly.

# Towers of Hanoi

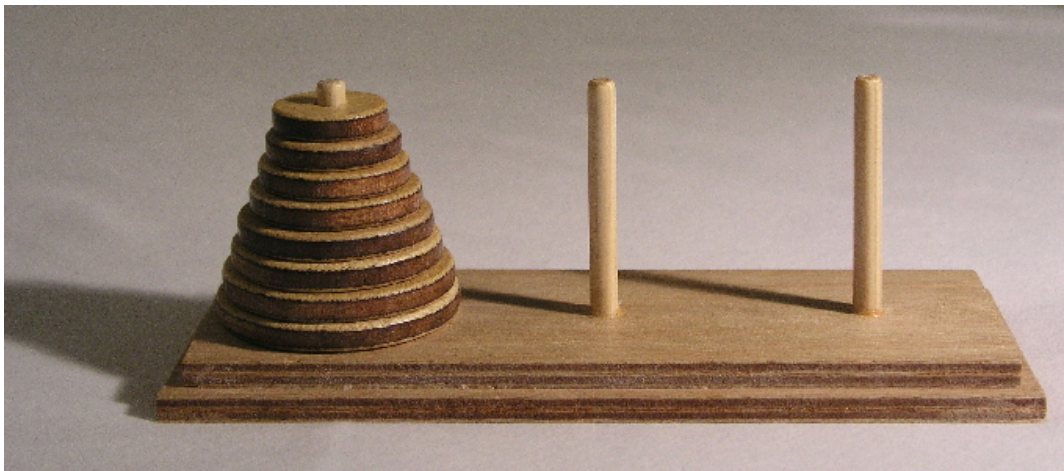
[https://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](https://en.wikipedia.org/wiki/Tower_of_Hanoi)

- ▶ The Tower of Hanoi is a mathematical game or puzzle.
- ▶ It consists of three rods and a number of disks of different sizes, which can slide onto any rod.
- ▶ The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top.
- ▶ The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
  1. Only one disk can be moved at a time.
  2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
  3. No larger disk may be placed on top of a smaller disk.
  4. With 3 disks, the puzzle can be solved in 7 moves.

The minimal number of moves required to solve a Tower of Hanoi puzzle is  $2^n - 1$ , where  $n$  is the number of disks.

# Towers of Hanoi

Let's watch the animation at [https://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](https://en.wikipedia.org/wiki/Tower_of_Hanoi)



# Iterative solution

- ▶ A simple solution for the toy puzzle is to **alternate moves between the smallest piece and a non-smallest piece**.
- ▶ When **moving the smallest piece**, always move it to the next position in the same direction (to the right if the starting number of pieces is even, to the left if the starting number of pieces is odd).
- ▶ If there is no tower position in the chosen direction, move the piece to the opposite end, but then continue to move in the correct direction.
- ▶ For example, if you started with three pieces, you would move the smallest piece to the opposite end, then continue in the left direction after that.
- ▶ When the turn is to move the non-smallest piece, there is only one legal move.
- ▶ Doing this will complete the puzzle in the fewest moves.



# Code

demo/example20.py

```
def moveTower(height,fromPole, toPole, withPole):  
    if height >= 1:  
        moveTower(height-1,fromPole,withPole,toPole)  
        moveDisk(fromPole,toPole)  
        moveTower(height-1,withPole,toPole,fromPole)  
  
def moveDisk(fp,tp):  
    print("moving disk from",fp,"to",tp)  
  
no_of_disks = 5  
moveTower(no_of_disks,"A","B","C")
```



# Towers of Hanoi — Complexity

Let  $t_n$  denote time to solve tower

$$\begin{aligned}t_n &= 2t_{n-1} + 1 \\&= 2(2t_{n-2} + 1) + 1 \\&= 4t_{n-2} + 2 + 1 \\&= 4(2t_{n-3} + 1) + 2 + 1 \\&= 8t_{n-3} + 4 + 2 + 1 \\&= 2^k t_{n-k} + 2^{k-1} + \dots + 4 + 2 + 1 \\&= 2^{n-1} + 2^{n-2} + \dots + 4 + 2 + 1 \\&= 2^n - 1\end{aligned}$$

Geometric growth

$$a = 2^{n-1} + \dots + 2 + 1$$

$$2a = 2^n + 2^{n-1} + \dots + 2$$

$$a = 2^n - 1$$

so order of growth is  $O(2^n)$

# Example: Exponential Complexity

- ▶ Given a set of integers (with no repeats), **want to generate the collection of all possible subsets** — called the power set.
- ▶  $\{1, 2, 3, 4\}$  **would generate**

$\{\}, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\},$   
 $\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}$

- ▶ **Order doesn't matter** — same set as well

·  $\{\}, \{1\}, \{2\}, \{1, 2\}, \{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}, \{4\}, \{1, 4\},$   
 $\{2, 4\}, \{1, 2, 4\}, \{3, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}$

# Power Set Concept — Recursively

- ▶ We want to generate the **power set of integers from 1 to  $n$**
- ▶ Assume we can generate **power set of integers from 1 to  $n - 1$** .
- ▶ Then all of those subsets belong to bigger power set (choosing not include  $n$ ); and all of those subsets with  $n$  added to each of them also belong to the bigger power set (choosing to include  $n$ ).
- ▶  $\{\}, \{1\}, \{2\}, \{1, 2\}, \{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}, \{4\}, \{1, 4\}, \{2, 4\}, \{1, 2, 4\}, \{3, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}$
- ▶ Nice recursive description!

# Exponential Complexity

demo/example21.py

```
def genSubsets(L):  
    res = []  
    if len(L) == 0:  
        return [[]] ##list of empty sets  
    smaller = genSubsets(L[:-1]) #all subsets without last element  
    extra = L[-1:]  
    new = []  
    for small in smaller:  
        new.append(small + extra) #for all smaller sol, add one with last el.  
    return smaller + new  
  
Ltest = [1,2,3,4,5]  
print(genSubsets(Ltest))
```

# Exponential Complexity (II)

```
def genSubsets(L):  
    res = []  
    if len(L) == 0:  
        return [[]]  
    smaller = genSubsets(L[:-1])  
    extra = L[-1:]  
    new = []  
    for small in smaller:  
        new.append(small + extra)  
    return smaller + new
```

- ▶ assuming append is constant time
- ▶ time includes time to solve smaller problem, plus time needed to make a copy of all elements in smaller problem

# Exponential Complexity (III)

```
def genSubsets(L):  
    res = []  
    if len(L) == 0:  
        return [[]]  
    smaller = genSubsets(L[:-1])  
    extra = L[-1:]  
    new = []  
    for small in smaller:  
        new.append(small + extra)  
    return smaller + new
```

- ▶ but important to think about size of smaller
- ▶ know that for a set of size  $k$  there are  $2^k$  cases
- ▶ how can we deduce overall complexity?

# Exponential Complexity (IV)

- ▶ let  $t_n$  denote time to solve problem of size  $n$
- ▶ let  $s_n$  denote size of solution for problem of size  $n$
- ▶  $t_n = t_{n-1} + s_{n-1} + c$  (where  $c$  is some constant number of operations)

$$t_n = t_{n-1} + 2^{n-1} + c$$

$$= t_{n-2} + 2^{n-2} + c + 2^{n-1} + c$$

$$= t_{n-k} + 2^{n-k} + \dots + 2^{n-1} + kc$$

$$= t_0 + 2^0 + \dots + 2^{n-1} + nc$$

$$= 1 + 2^n + nc$$

Thus computing power set is  $O(2^n)$



# Analyze Iterative Fibonacci for Complexity

demo/example22.py, demo/example21.py

```
def fib_iter(n):
```

```
    if n == 0:
```

```
        return 0
```

```
    elif n == 1:
```

```
        return 1
```

```
    else:
```

```
        fib_i = 0
```

```
        fib_ii = 1
```

```
        for i in range(n-1):
```

```
            tmp = fib_i
```

```
            fib_i = fib_ii
```

```
            fib_ii = tmp + fib_ii
```

```
        return fib_ii
```

constant  
 $O(1)$

constant  
 $O(1)$

linear  
 $O(n)$

constant  
 $O(1)$

- Best case:

$O(1)$

- Worst case:

$O(1) + O(n) + O(1) \rightarrow \mathbf{O(n)}$

$$F_0 = 0, \quad F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2}, \quad \text{for } n > 1.$$

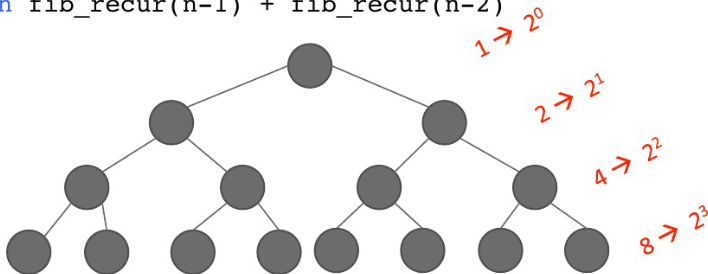
# Analyze Recursive Fibonacci for Complexity

demo/example23.py

```
def fib_recur(n):  
    """ assumes n an int >= 0 """  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib_recur(n-1) + fib_recur(n-2)
```

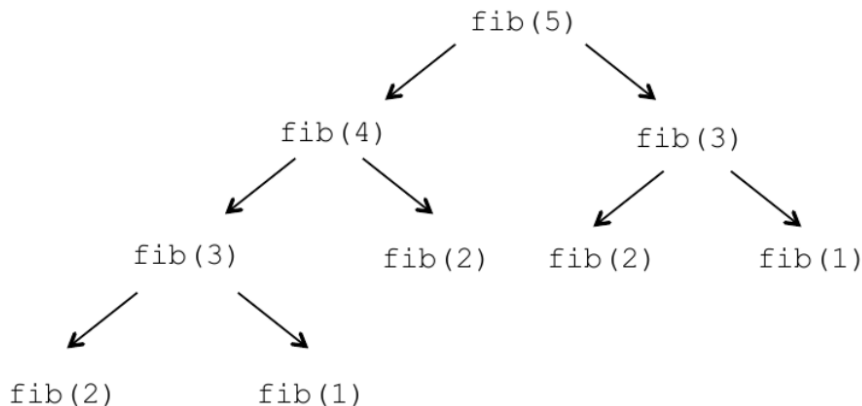
- Worst case:

$O(2^n)$



# Analyze recursive Fibonacci for Complexity (II)

- ▶ Actually can do a bit better than  $2^n$  since tree of cases thins out to right.
- ▶ But complexity is still exponential.



# Complexity of some Python functions

Lists:  $n$  is `len(L)`

- ▶ `index` —  $O(1)$
- ▶ `store` —  $O(1)$
- ▶ `length` —  $O(1)$
- ▶ `append` —  $O(1)$
- ▶ `==` —  $O(n)$
- ▶ `remove` —  $O(n)$
- ▶ `copy` —  $O(n)$
- ▶ `reverse` —  $O(n)$
- ▶ `iteration` —  $O(n)$
- ▶ `in list` —  $O(n)$

Dictionaries:  $n$  is `len(L)`

Worst case:

- ▶ `index` —  $O(n)$
- ▶ `store` —  $O(n)$
- ▶ `length` —  $O(n)$
- ▶ `delete` —  $O(n)$
- ▶ `iteration` —  $O(n)$

Average case:

- ▶ `index` —  $O(1)$
- ▶ `store` —  $O(1)$
- ▶ `delete` —  $O(1)$
- ▶ `iteration` —  $O(n)$

### 3. Some useful Libraries in Python

1. Numpy
2. Scipy
3. Matplotlib
4. Pandas
5. JAX

NumPy, SciPy, Matplotlib, Pandas, JAX...

## Top Python Data Science Libraries



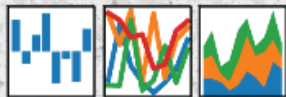
NumPy

matplotlib



pandas

$$y_i = \beta' x_i + \mu_i + \epsilon_i$$



# Python and Libraries

If we use Python in combination with its modules

- ▶ NumPy
- ▶ SciPy
- ▶ Matplotlib
- ▶ Pandas
- ▶ JAX

it belongs to the top numerical programming languages.



# Do not re-invent the wheel — NumPy

**NumPy** is a package for linear algebra and advanced mathematics in Python.

It provides a *fast* implementation of multidimensional numerical arrays (C/FORTRAN like), vectors, matrices, tensors and operations on them.

*Use it if:* you long for MATLAB core features.

*See also:* <http://www.numpy.org/>



# Do not re-invent the wheel — SciPy

“**SciPy** is open-source software for mathematics, science, and engineering. [...] The SciPy library provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization.”

One of its main aim is to provide a reimplementation of the MATLAB toolboxes.

*Use it if:* you long for MATLAB toolbox features.

*See also:* <http://www.scipy.org/>

# Do not re-invent the wheel — Matplotlib

**matplotlib** “is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits.”

# Pandas

**Pandas** is a Python data analysis library, that provides optimized routines for analyzing 2D, 3D, 4D data.

“Pandas [...] enables you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language like R.”

*Use it if:* you need features from R, `plyr`, `reshape2`.

# Simple Examples

Let's have a look at the Jupyter Notebook

**Lecture\_5b.ipynb**

# Numpy Example: Poisson equation

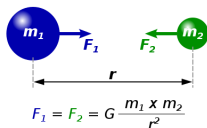
- Consider

$$\nabla^2 u = \Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f, \mathbf{x} \in \Omega \text{ Domain}$$

- with boundary condition

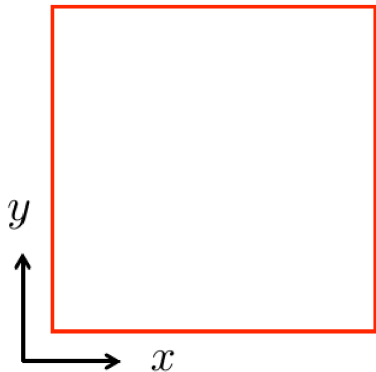
$$u = g, \mathbf{x} \in \partial\Omega \text{ Boundary of domain}$$

- Model useful in: heat conduction, electromagnetism, astrophysics (gravity), fluid dynamics, ...
- Simplest example of an elliptic Partial Differential Equation (PDE)



# Numpy Example: Poisson equation

- ▶ Consider  $\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f$
- ▶ on the square domain

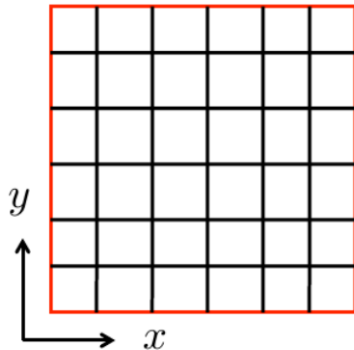


# Example: Poisson equation (III)

- Finite differences

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f$$

- discretize square domain



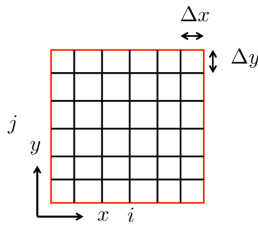
# Example: Poisson equation (IV)

- Finite differences

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f$$

$$\begin{aligned}\nabla^2 u &\approx \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j))}{\Delta x^2} \\ &\quad + \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1}))}{\Delta y^2} \\ &= f(u(x_i, y_j)).\end{aligned}$$

- discretize square domain



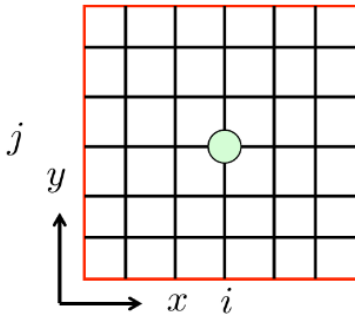


# Example: Poisson equation (V)

- Finite differences

$$\nabla^2 u \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = f_{i,j}$$

- discretize square domain



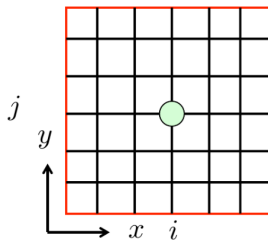
# Jacobi-Method

- Jacobi-method

$$u_{i,j}^{n+1} = \frac{1}{2(\Delta x^2 + \Delta y^2)} [(u_{i+1,j}^n + u_{i-1,j}^n) \Delta y^2 + (u_{i,j+1}^n + u_{i,j-1}^n) \Delta x^2 - f_{i,j} \Delta x^2 \Delta y^2]$$

- Discretized square domain

*iterate ...*



# Poisson Equation

$$u_{i,j}^{n+1} = \frac{1}{2(\Delta x^2 + \Delta y^2)} [(u_{i+1,j}^n + u_{i-1,j}^n) \Delta y^2 + (u_{i,j+1}^n + u_{i,j-1}^n) \Delta x^2 - f_{i,j} \Delta x^2 \Delta y^2]$$

```
def update(u,dx,dy):  
    [nx,ny] = u.shape  
    dx2 = dx**2  
    dy2 = dy**2  
    u_old = np.copy(u)  
    for i in range(1,nx-1):  
        for j in range(1,ny-1):  
            u[i,j] = ( (u_old[i+1,j] + u_old[i-1,j]) * dy2 \  
                      + (u_old[i,j+1] + u_old[i,j-1]) * dx2) \  
                      / (2 * (dx2 + dy2))
```

# Poisson Equation

Compare implementations (vectorized):

$$u_{i,j}^{n+1} = \frac{1}{2(\Delta x^2 + \Delta y^2)} [(u_{i+1,j}^n + u_{i-1,j}^n) \Delta y^2 + (u_{i,j+1}^n + u_{i,j-1}^n) \Delta x^2 - f_{i,j} \Delta x^2 \Delta y^2]$$

```
def update(u,dx,dy):  
    dx2 = dx**2  
    dy2 = dy**2  
    u_old = np.copy(u)  
    u[1:-1,1:-1] = ( (u_old[2:,1:-1] + u_old[:-2,1:-1])*dy2 \  
                    + (u_old[1:-1,2:] + u_old[1:-1,:-2])*dx2) \  
                    / (2*(dx2 + dy2))
```

# Nonlinear equations & optimization.

- ▶ Our course heavily relies on solving large **systems of nonlinear equations** or (un-)constrained **optimization problems**.
- ▶ → In Python, you have plenty of options, e.g.:
- ▶ SciPy.org
- ▶ PyOpt.org
- ▶ IPOPT (<https://www.coin-or.org/lpopt>;  
<https://github.com/xuy/pyipopt>)

# Constrained optimization with SciPy

The minimize function also provides an interface to several constrained minimization algorithm.

As an example, the Sequential Least Squares Programming optimization algorithm (SLSQP) will be considered here.

This algorithm allows to deal with constrained minimization problems of the form:

$$\begin{aligned} \min F(x) \\ \text{subject to } C_j(X) = 0, \quad j = 1, \dots, \text{MEQ} \\ C_j(x) \geq 0, \quad j = \text{MEQ} + 1, \dots, M \\ XL \leq x \leq XU, l = 1, \dots, N. \end{aligned}$$

# Constrained optimization — example

As an example, let us consider the problem of optimizing the function:

$$f(x, y) = 2xy + 2x \cdot x^2 \cdot 2y^2$$

subject to an equality and an inequality constraints defined as:

$$\begin{aligned}x^3 - y &= 0 \\ y - 1 &> 0\end{aligned}$$

# Root finding (nonlinear equations)

- ▶ Finding a root of a set of non-linear equations can be achieved using the root function.
- ▶ Several methods are available, amongst which `hybr` (the default) and `lm` which respectively use the hybrid method of Powell and the LevenbergMarquardt method from MINPACK.
- ▶ Consider a set of non-linear equations

$$x_0 \cos(x_1) = 4,$$

$$x_0 x_1 - x_1 = 5.$$



# Example for Nonlinear Equations

```
import numpy as np
from scipy.optimize import root

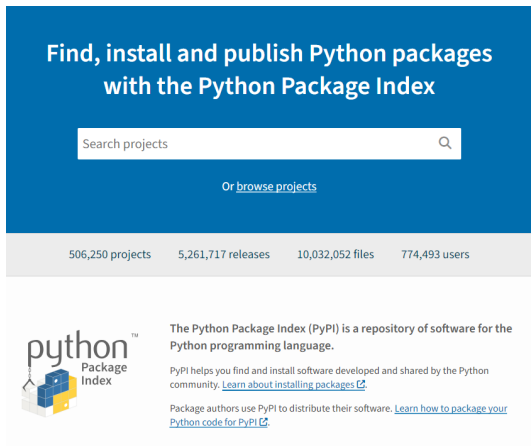
def func2(x):
    f = [x[0] * np.cos(x[1]) - 4, x[1]*x[0] - x[1] - 5]
    df = np.array([[np.cos(x[1]), -x[0] * np.sin(x[1])], [x[1], x[0] - 1]])
    return f, df

sol = root(func2, [1, 1], jac=True, method='lm')
solution = sol.x

print("the solution of this nonlinear set of equations is: ", solution)
```

# Want more?

PyPI is the index of Python software packages. It currently indexes 506,250 packages, so the choice is really vast. Almost all packages can be installed with a single command by running `pip install packagename`.




The screenshot shows the PyPI homepage with a blue header and a white search bar. The search bar contains the text "Search projects" and a magnifying glass icon. Below the search bar is a link "Or browse projects". A grey bar displays statistics: 506,250 projects, 5,261,717 releases, 10,032,052 files, and 774,493 users. The footer features the PyPI logo and a description of the index, along with links for learning more about installing packages and packaging code for PyPI.

Find, install and publish Python packages with the Python Package Index

Search projects

Or [browse projects](#)

506,250 projects   5,261,717 releases   10,032,052 files   774,493 users

 The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. [Learn about installing packages](#)

Package authors use PyPI to distribute their software. [Learn how to package your Python code for PyPI](#)

# How to tackle a complete project in python

- ▶ The examples so far were **quite compact** and composed to convey programming constructs in a gentle pedagogical way.
- ▶ Now, the idea is to **solve a more comprehensive real-world problem by programming.**
- ▶ The problem solving process in this example gets quite involved.
- ▶ How to proceed:
  1. Problem Statement
  2. Derivation of the Algorithm
  3. Program Development and Testing
  4. Verification
  5. Visualization

Questions for today?

