

Introduction to Programming with Scientific Applications

Gerth Stølting Brodal
Department of Computer Science
Aarhus University



Course evaluation

*"The first lecture was intimidating
and overwhelming"*

Lecturer

Name	Gerth Stølting Brodal
Research	Algorithms and Data Structures (Computer Science)
Teaching	
2018 -	BSc course on Introduction to Programming with Scientific Applications
2003 -	BSc course on Introduction to Algorithms and Data Structures
1999 - 17	MSc courses on Computational Geometry, Algorithm Engineering, Advanced Data Structures, External Memory Algorithms and Data Structures
Python	Advanced Beginner

Course description – kursuskatalog.au.dk/en/course/130939/

Introduction to Programming with Scientific Applications

Description of qualifications

After the course the participants will have knowledge of principles and techniques for systematic **construction** of **programs**.

At the end of the course, the participants will be able to:

- apply constructions of a common programming language,
- develop **well-structured** programs and perform **testing** and **debugging** of these,
- explain fundamental programming concepts and basic algorithmic techniques,
- apply standard **tools for scientific applications**,
- use the documentation for a programming language and available software packages.

Contents

The course gives an introduction to programming with scientific applications.

Programming concepts and techniques are introduced using the **Python** programming language.

The programming concepts are **illustrated in other programming languages**. The following content is included.

Basic programming constructs: Data types, operators, variables, flow of control, conditionals, loops, functions, recursion, scope, exceptions. *Object orientation:* Abstract data types, classes, inheritance, encapsulation. *Basic algorithmic techniques:* Sorting, binary search, dynamic programming. *Systematic development of programs:* Testing and debugging. File-based input/output, numerical analysis, functional programming. Scientific computing using standard packages for Python.

ECTS 10

Hours - weeks - periods

Lectures 2 x 2 hours/week

TA sessions 1 x 3 hours/week

Study café 3 x 1 hour/week

Language of instruction

Danish

Instructor

Gerth Stølting Brodal

Academic prerequisites

(Some) Linear algebra

Exam

5 hour programming

Aid: Computer and Internet, headphones, no AI

7-point grading scale

Prerequisites for examination participation

Submission and approval of 10 mandatory assignments and submission of

1 implementation project

Notes Grade reflects an overall assessment of implementation project and written examination. Project counts 20% and written exam counts 80%

Question – Primary Education?

- a) Mathematics
- b) Mathematics-Economics
- c) Data Science
- d) Chemistry
- e) Physics
- f) Other Science-Technology
- g) Other

Question – Programming languages you know?

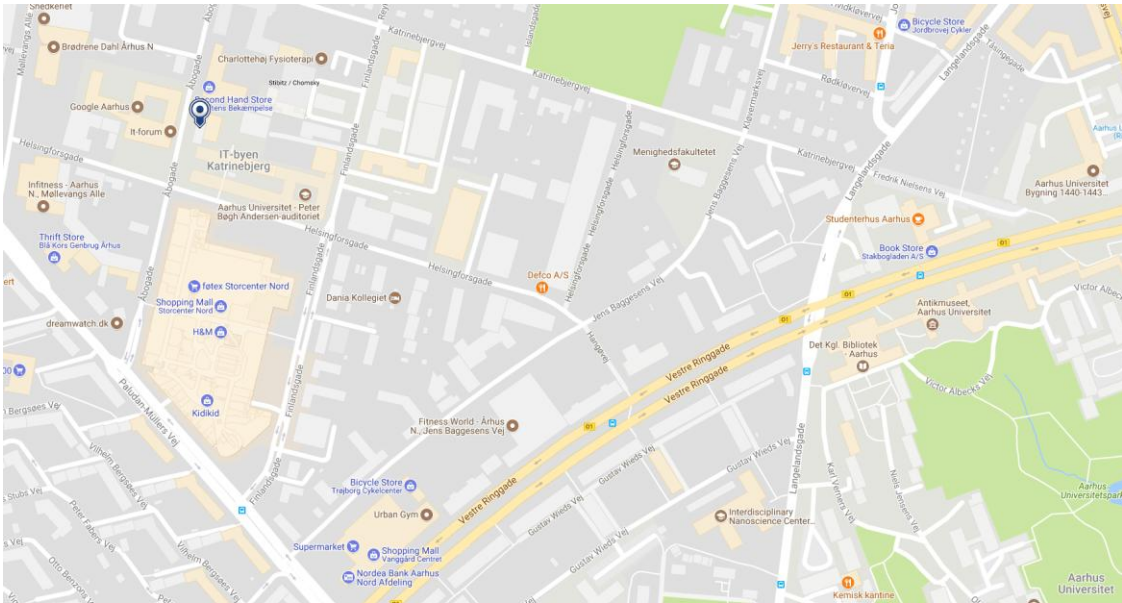
Question – Programming experience?

For the programming language you know best (if any) please state your proficiency level within the language.

- a) None
- b) Fundamental awareness (basic knowledge)
- c) Novice (limited experience)
- d) Intermediate (practical application)
- e) Advanced (applied theory)
- f) Expert (recognized authority)

Some course practicalities

Primary lecture material = slides



Week	Monday	Tuesday	Wednesday	Thursday	Friday
5	F1	no TA class	F2		
6	F3	TØ1	TØ1 / F4	TØ1	TØ1
7	F5	TØ2	TØ2 / F6	TØ2	TØ2
8	F7	TØ3	TØ3 / F8	TØ3	TØ3
9	F9	TØ4	TØ4 / F10	TØ4	TØ4
10	F11	TØ5	TØ5 / F12	TØ5	TØ5
11	F13	TØ6	TØ6 / F14	TØ6	TØ6
12	F15	TØ7	TØ7 / F16	TØ7	TØ7
13	F17	TØ8	TØ8 / F18	TØ8	TØ8
14	F19	TØ9	TØ9 / F20	TØ9	TØ9
15	F21	TØ10	TØ10 / F22	TØ10	TØ10
16	Easter break				
17		-	-	-	Kapsejlads?
18	F23	TØ11	TØ11 / F24	TØ11	TØ11
19	F25	TØ12	TØ12 / F26	TØ12	TØ12
20	F27	TØ13	TØ13 / -	TØ13	TØ13

	Monday	Tuesday	Wednesday	Thursday	Friday
8:15-9:00	TA meeting				
9:15-10:00	Study cafe		Study cafe	MA1 (1Y)	
10:15-11:00	Lecture		Lecture		MA2 (1Y)
11:15-12:00					
12:15-13:00					
13:15-14:00			DV		
14:15-15:00		MA3 (2Y)	MØ1	Study cafe	Hold 2
15:15-16:00		FY	MØ2		
16:15-17:00		Hold 1			
17:15-18:00					

Course page on Brightspace and GitHub

 Introduktion til programmering med videnskabelige an...



Gerth Støtting Brodal
as Student

Course Home Content Course Tools ▾ Classlist Zoom Panopto Help

gsbrodal.github.io/ipsa

Course information

Who, where and when

Handin deadlines

Course content on GitHub

Course information

Welcome to the course *Introduction to Programming with Scientific Applications*

The information about the course is available on the Brightspace page and a public GitHub repository. You can use the discussion forum for questions, including the course content on GitHub.



Introduction to Programming with Scientific Applications

Aarhus University, Department of Computer Science

Welcome

Course plan

- Compact
- All slides (pdf)

Exercises

Handins

Final project

- I - Medians
- II - Portfolio
- III - NMR
- IV - MNIST
- V - Open topic

Exam

- Statistics
- Past exams

Plagiarism

Workload

Literature

Python installation

- Windows 11
- Mac & Linux

Python resources

- Books
- Videos

AU course description

Welcome

This page contains the public content for the course *Introduction to Programming with Scientific Applications* offered by Aarhus University, Department of Computer Science. The lecturer is [Gerth Støtting Brodal](#).

Class lists, discussions, student handins and feedback are handled using Aarhus University's learning management system Brightspace at [brightspace.au.dk](#).

The course gives an introduction to the Python 3 programming language and applications using Python. Throughout the course students are encouraged to seek online information in e.g. the Python language specification.

The course will be run with weekly 2 x 2 hours lectures (alternatively recorded lectures on YouTube), 3 hours of exercise classes with a teaching assistant ("øvelser"), and 3 hours of staffed study café.

During the course students are required to hand in 10 weekly handins and one larger implementation project. Handins and the project are done in groups of up to three persons. Approval of the weekly handins is a prerequisite to attend the exam. The final exam will be a programming exam with all aids, incl. internet, and *the final grade will be based on an overall evaluation of the implementation project (20%) and the programming exam (80%)*.

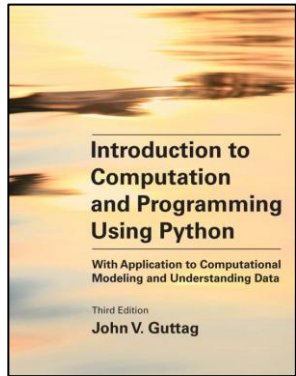
Course content

The course gives an introduction to programming with scientific applications. Programming concepts and techniques are introduced using the Python programming language. The programming concepts are illustrated in other programming languages. The following content is included.

- Basic programming constructs: Data types, operators, variables, flow of control, conditionals, loops, functions, recursion, scope, exceptions.

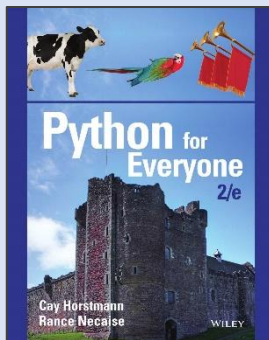


Course text book – optional



John V. Guttag: **Introduction to Computation and Programming Using Python, Third Edition With Application to Computational Modeling and Understanding Data**. Third Edition. 664 pages. MIT Press, 2021.

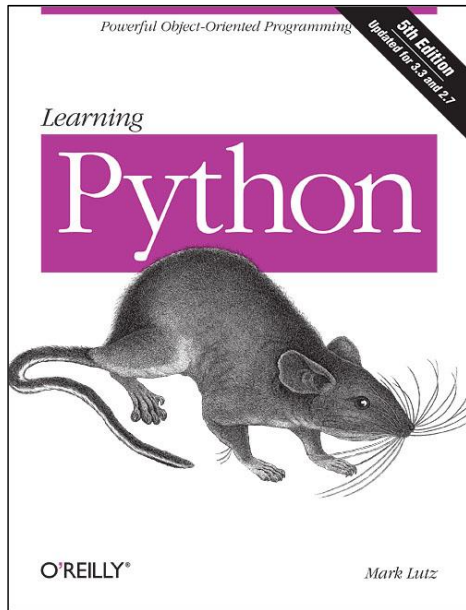
- *[Guttag, 2nd Ed., page 8] "The reader should be forewarned that this book is by no means a comprehensive introduction to Python". 3rd Ed. added about 80 pages on introduction to Python.*
- *Covers all basic features of Python enabling you to deal with data in Chapters 1-10 (212 pages) - remaining chapters are applications*
- *Other resources: Google, stackoverflow, Python.org, YouTube, Als...*



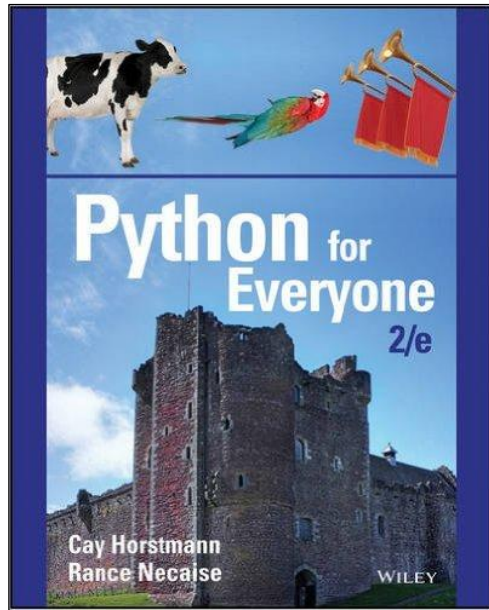
Comparison to a standard text book on the *programming language* Python by Cay Horstmann and Rance Necaise:

Topic **recursion** is covered by Guttag on page 123 (2nd edition on page 50), Horstmann and Necaise do it on page 611

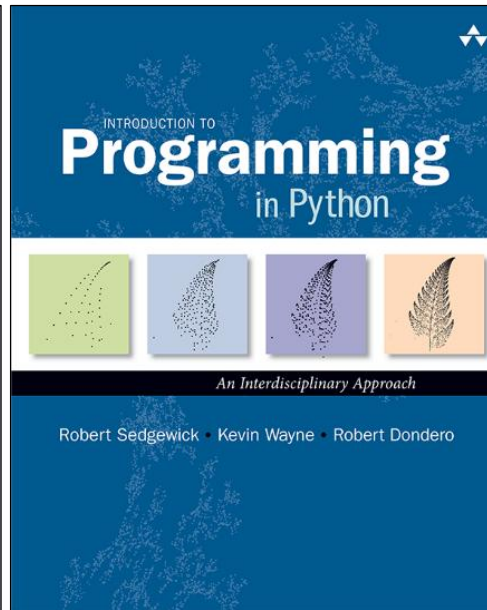
Some other books on Python



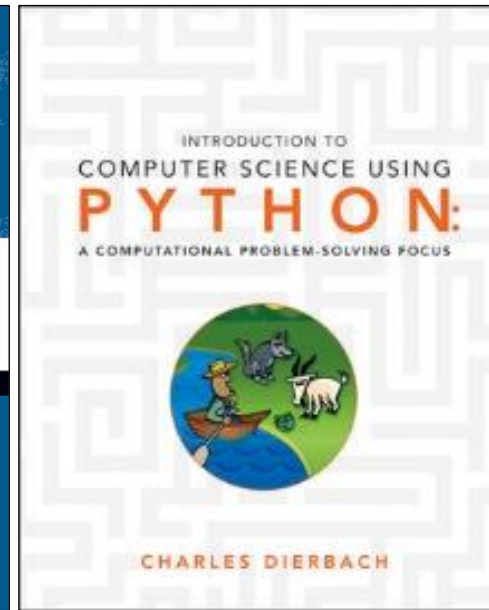
O'Reilly, 2013
1684 pages



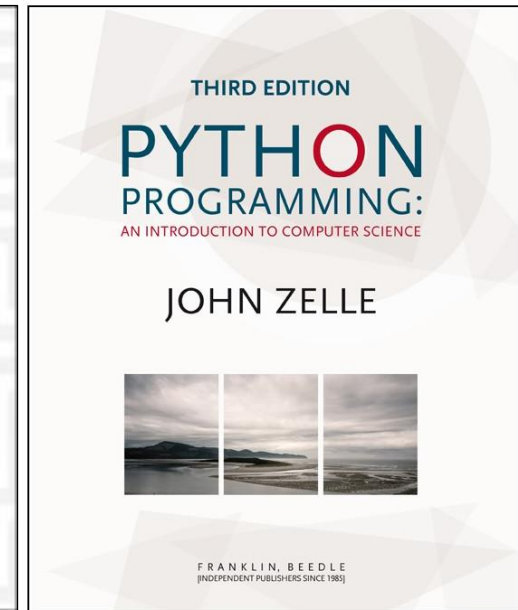
Wiley, 2016
752 pages



Addison-Wesley, 2015
794 pages



Wiley, 2013
580 pages



Franklin & Beedle, 2016
552 pages

... numerous online introduction texts/courses/videos on Python

Two Python programs

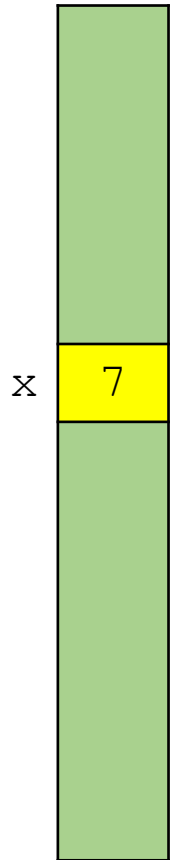
A Python program

Python shell

```
> x = 7
> print(x * x)
| 49
```

- 7 is an *integer literal* – in Python denoted an “int”
- x is the name of a *variable* that can hold some value
- = is assigning a value to a variable
- * denotes multiplication
- print is the name of a built-in *function*,
here we call print to print the result of 7 * 7
- A program consists of a sequence of *statements*, executed sequentially

Memory



Question – What is the result of this program?

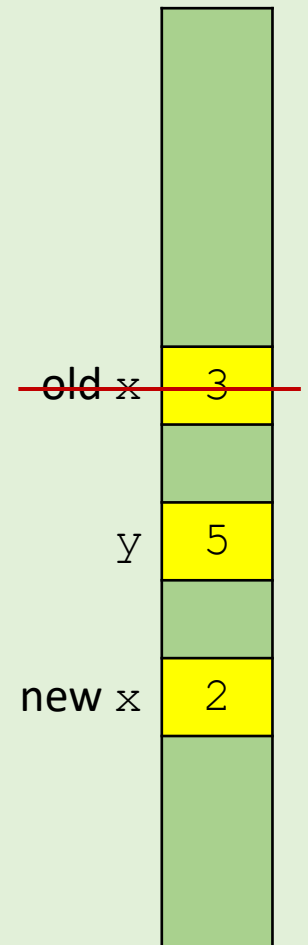
Python shell

```
> x = 3  
> y = 5  
> x = 2  
> print(x * y)
```

x assigned new value

- 😊 a) 10
- b) 15
- c) 25
- d) [15, 10]
- e) Error
- f) Don't know

Memory

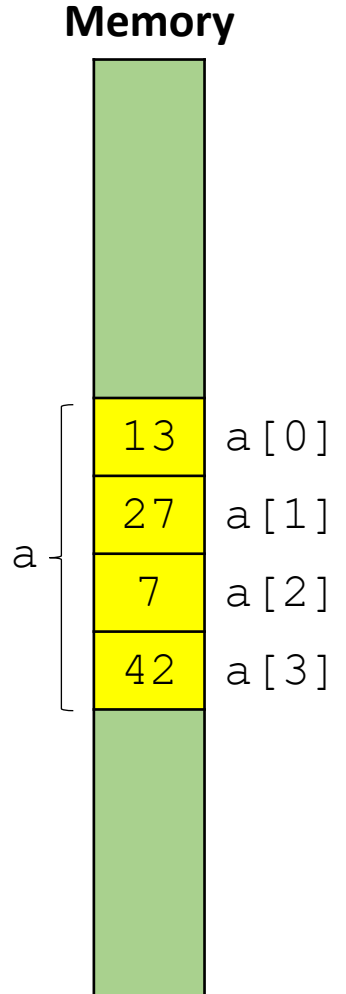


Another Python program using lists

Python shell

```
> a = [13, 27, 7, 42]
> print(a)
| [13, 27, 7, 42]
> print(a[2])
| 7
```

- `[13, 27, 7, 42]` is a *list* containing four integers
- `a[2]` refers to the entry in the list with *index* 2 (the first element has index 0, i.e. `a[2]` is the 3rd element of the list)
- Note that `print` also can print a list

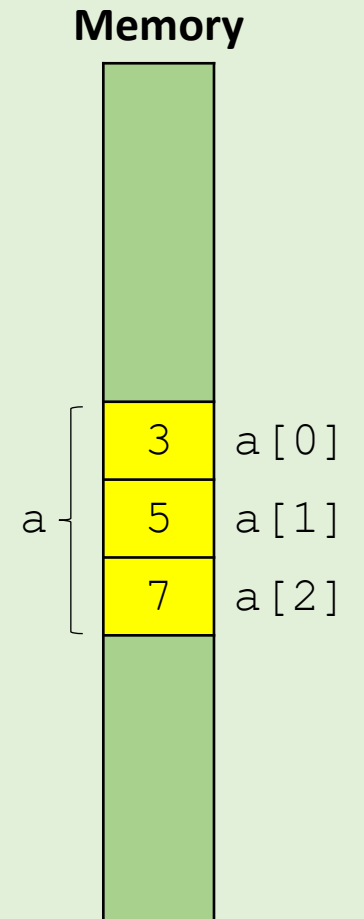


Question – What is the result of this program?

Python shell

```
> a = [3, 5, 7]  
> print(a[1] + a[2])
```

- a) 8
- b) 10
- 😊 c) 12
- d) 15
- e) Don't know



Why Python ?



the next slides will be technical

TIOBE Index January 2025

Python #1
Since
November
2021

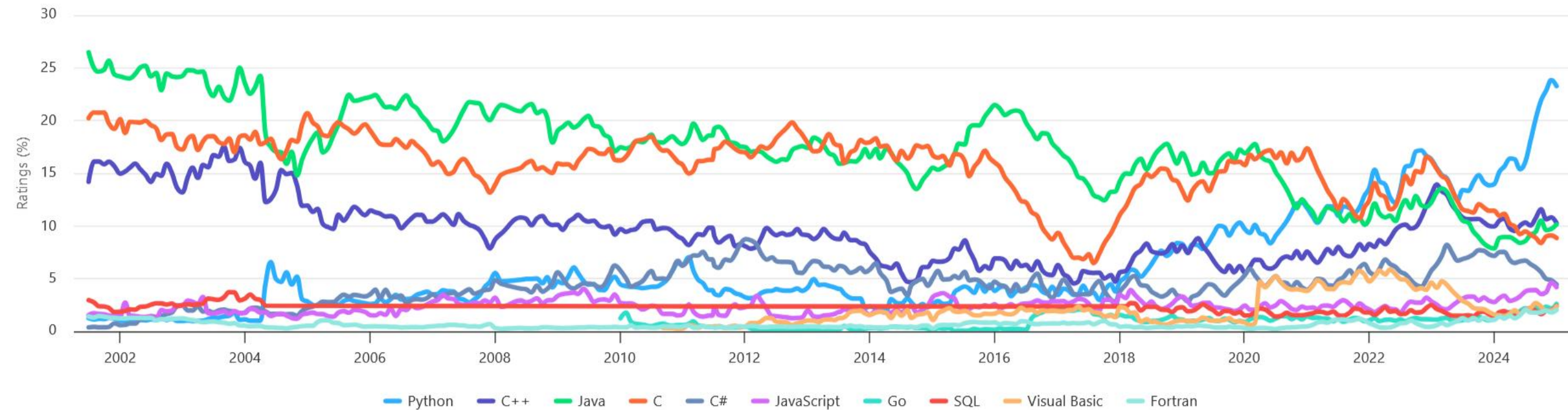
Jan 2025	Jan 2024	Change	Programming Language		Ratings	
1	1			Python	23.28%	+9.32%
2	3	^		C++	10.29%	+0.33%
3	4	^		Java	10.15%	+2.28%
4	2	v		C	8.86%	-2.59%
5	5			C#	4.45%	-2.71%
6	6			JavaScript	4.20%	+1.43%
7	11	^^		Go	2.61%	+1.24%
8	9	^		SQL	2.41%	+0.95%
9	8	v		Visual Basic	2.37%	+0.77%
10	12	^		Fortran	2.04%	+0.94%

The TIOBE Programming Community index is an indicator of the **popularity of programming languages**. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

Popularity of programming languages

TIOBE Programming Community Index

Source: www.tiobe.com



“Hello World”

- In Java, C, C++ a lot of “{”, “}” and “;” are needed
- Java tends to have a lot of “public...” details that need to be spelled out
- Python is concise

Java

```
public class HelloWorld {  
    public static void main( String[] args ) {  
        System.out.println( "Hello World!" );  
        System.exit( 0 );  
    }  
}
```

C

```
#include <stdio.h>  
  
int main(int argc, char **argv) {  
    printf("Hello World");  
    return 0;  
}
```

C++

```
#include <iostream>  
using namespace std;  
  
int main(int argc, char** argv) {  
    cout << "Hello, World!";  
    return 0;  
}
```

Python 2

```
print "Hello world"
```

Python 3

```
print("Hello world")
```

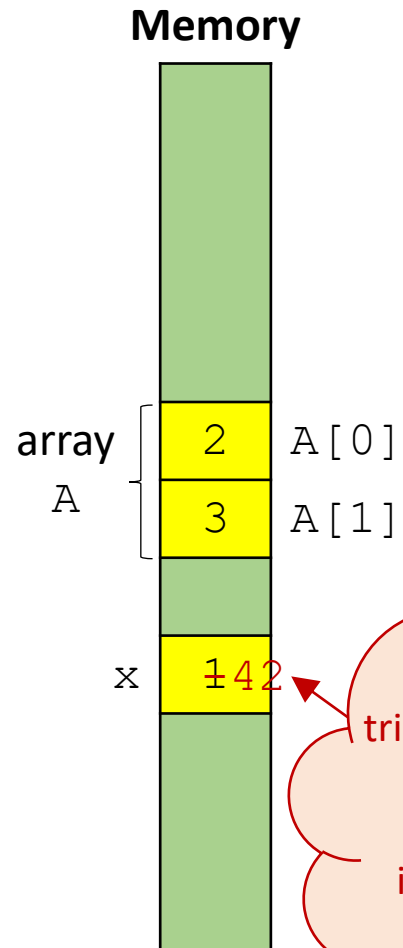

Why Python ?

- **Short concise code**

C index out of bounds

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

en.wikipedia.org/wiki/Debugging



“A” only has size 2, but tries to update the 4th entry. No warning is giving. Something unexpected is overridden in memory. **Have fun debugging!**

indexing.c

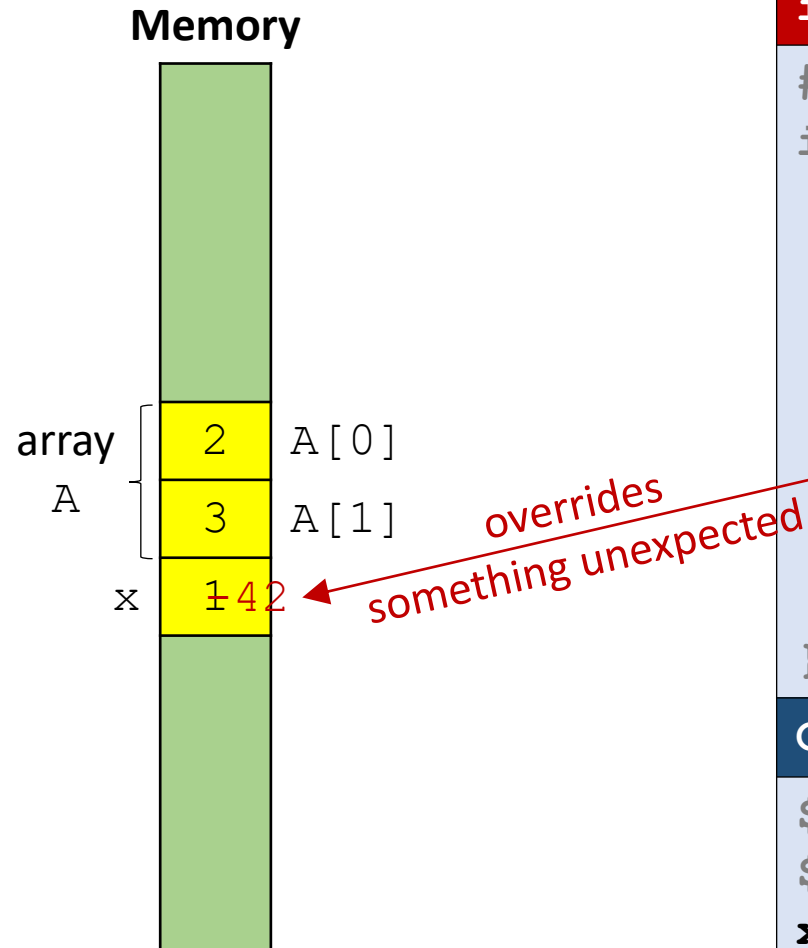
```
#include <stdio.h>
int main() {
    int x = 1;
    int A[2] = {2, 3}; // A[0] = 2, A[1] = 3
    printf("x = %d, A = {%d, %d}\n", x, A[0], A[1]);
    A[3] = 42; // index A[3] out of bounds
    printf("x = %d, A = {%d, %d}\n", x, A[0], A[1]);
    return 0;
}
```

Output

```
$ gcc indexing.c
$ ./a.exe
x = 1, A = {2, 3}
x = 42, A = {2, 3}
```

Skipping checking for invalid indexing makes programs faster, but also requires disciplined programming

... and C++ index out of bounds



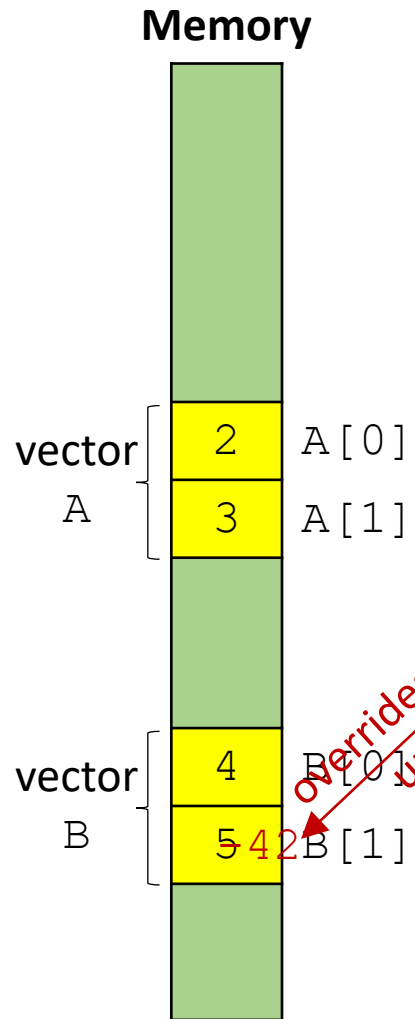
indexing.cpp

```
#include <iostream>
int main() {
    int x = 1;
    int A[2] = {2, 3}; // A[0] = 2, A[1] = 3
    std::cout << "x = " << x << ", A = {"
                << A[0] << ", " << A[1] << "}" << std::endl;
    A[2] = 42; // index A[2] out of bounds
    std::cout << "x = " << x << ", A = {"
                << A[0] << ", " << A[1] << "}" << std::endl;
    return 0;
}
```

Output

```
$ g++ indexing.cpp
$ ./a.exe
x = 1, A = {2, 3}
x = 42, A = {2, 3}
```

... and C++ vector index out of bounds



overrides something
unexpected

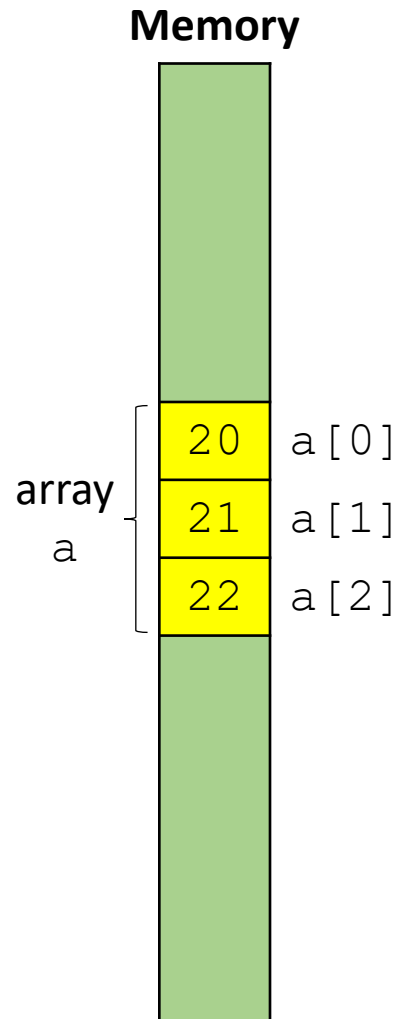
indexing.cpp

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> A = {2, 3}; // A[0] = 2, A[1] = 3
    std::vector<int> B = {4, 5}; // B[0] = 4, B[1] = 5
    std::cout << "A={" << A[0] << ", " << A[1] << "}, ";
    std::cout << "B={" << B[0] << ", " << B[1] << "}" << std::endl;
    A[9]=42; // index A[9] out of bounds
    std::cout << "A={" << A[0] << ", " << A[1] << "}, ";
    std::cout << "B={" << B[0] << ", " << B[1] << "}" << std::endl;
    return 0;
}
```

Output

```
$ g++ -std=c++11 indexing-vector.cpp
$ ./a.exe
A={2, 3}, B={4, 5}
A={2, 3}, B={4, 42}
```

... and Java index out of bounds exception



indexing.java

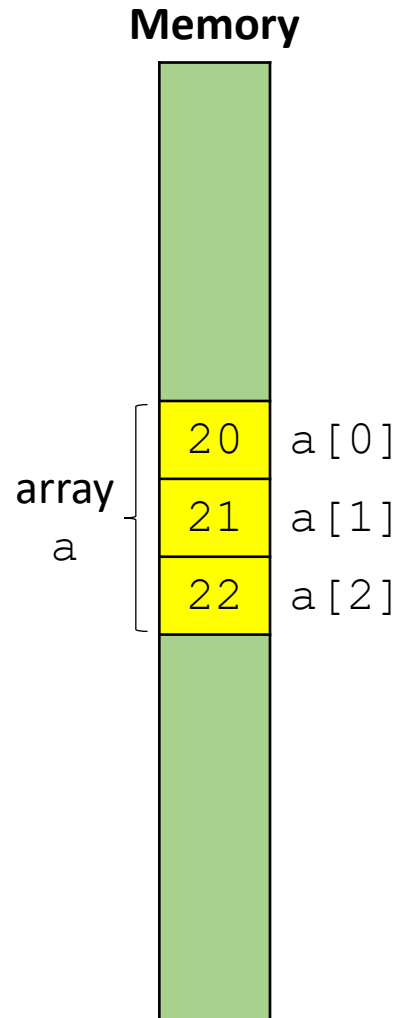
```
class IndexingTest{
    public static void main(String args[]){
        int a[] = {20, 21, 22};
        a[5] = 42; // index a[5] out of bounds
    }
}
```

Output

```
$ javac indexing.java
$ java IndexingTest
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
    at IndexingTest.main(indexing.java:5)
```

Java provides error message when running the program

... and Python index out of bounds exception



indexing.py

```
a = [20, 21, 22]
a[5] = 42 # index a[5] out of bounds
```

Output

```
$ python indexing.py
Traceback (most recent call last):
  File "indexing.py", line 3, in <module>
    a[5] = 42
IndexError: list assignment index out of range
```

Python provides error message when running the program

Memory safety

The White House 2024 | Press Release: “*Future Software Should Be Memory Safe*” (www.whitehouse.gov)

National Security Agency 2022 | Cybersecurity Information Sheet: *Software Memory Safety* (media.defense.gov)

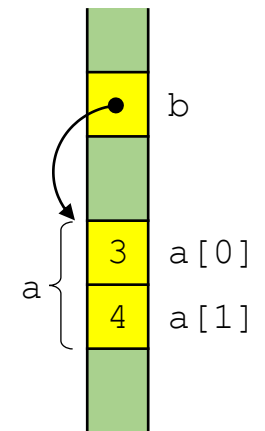
- C and C++ are flexible but **memory unsafe** programming languages
 - Unintended writes or reads to memory can be exploited by malicious cyber actors 
- Python, Java, Rust are examples of memory safe languages
- Rust aims at achieving the efficiency of C by slightly restricting flexibility

indexing.rs

```
fn main() {  
    let mut a = [3, 4];  
    a[2] = 7; // Compile error: this operation will panic at runtime  
    for i in 2..3 { a[i] = 7 } // Run-time panic: index out of bounds  
    let b = &mut a;  
    a[1] = 6; // Compile error: cannot use `a` because it was mutably borrowed  
    (*b)[0] = 5;  
    for i in 0..2 { println!("a[{}] = {}", i, a[i]) }  
}
```

www.rust-lang.org 

Memory



Why Python ?

- Short concise code
- **Index out-of-range exceptions**

C++ different ways to print a vector

vector-iterator.cpp

```
#include <iostream>
#include <vector>
int main() {
    // Vector is part of STL (Standard Template Library)
    std::vector<int> A = {20, 23, 26};
    // "C" indexing - since C++98
    for (int i = 0; i < A.size(); i++)
        std::cout << A[i] << std::endl;
    // iterator - since C++98
    for (std::vector<int>::iterator it = A.begin(); it != A.end(); ++it)
        std::cout << *it << std::endl;
    // "auto" iterator - since C++11
    for (auto it = A.begin(); it != A.end(); ++it)
        std::cout << *it << std::endl;
    // Range-based for-loop - since C++11
    for (auto e : A)
        std::cout << e << std::endl;
}
```

elegant

Java - different ways to print a vector

vector-iterator.java

```
import java.util.Vector;
import java.util.Iterator;

class IteratorTest{
    public static void main(String[] args) {
        Vector<Integer> a = new Vector<Integer>();
        a.add(7);
        a.add(42);
        // "C" for-loop & get method
        for (int i = 0; i < a.size(); i++)
            System.out.println(a.get(i));
        // iterator
        for (Iterator it = a.iterator(); it.hasNext(); )
            System.out.println(it.next());
        // for-each loop - since Java 5
        for (Integer e : a)
            System.out.println(e);
    }
}
```

elegant

The Python way to print a list

print-list.py

```
a = [20, 23, 26]
```

```
for e in a:  
    print(e)
```

Output

```
$ python print-list.py
```

```
20
```

```
23
```

```
26
```

Why Python ?

- Short concise code
- Index out of range exceptions
- **Elegant for-each loop**

C++ how not to print a vector

print-vector.cpp

```
#include <iostream>
#include <vector>

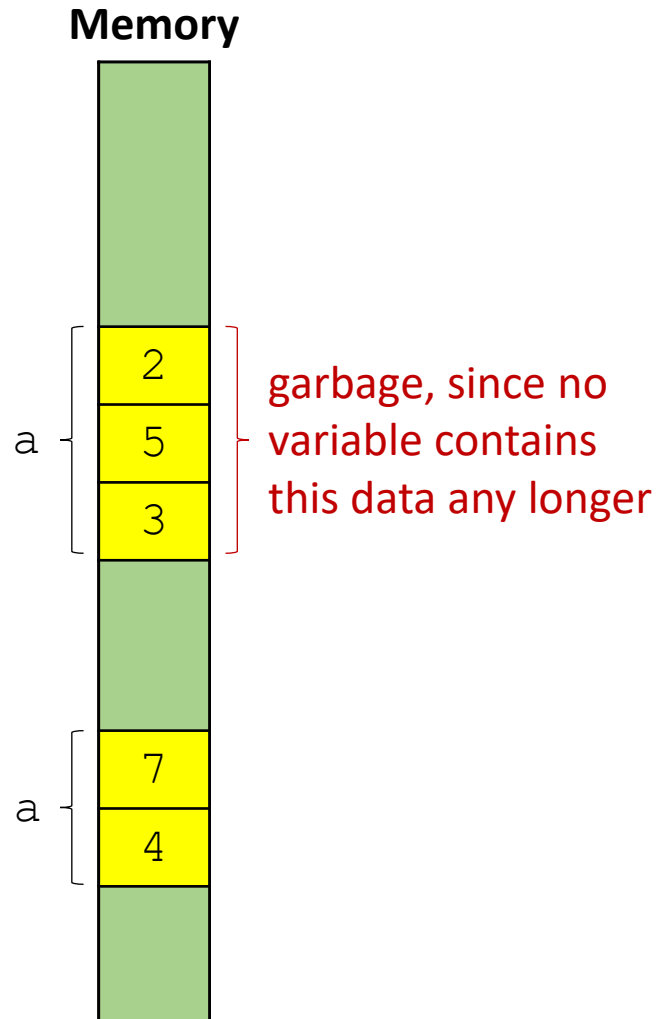
int main() {
    std::vector<int> A = {2, 3};
    std::cout << A << std::endl;
    return 0;
}
```

C++ vectors cannot be printed directly –
mistake results in +200 lines of error messages

Why Python ?

- Short concise code
- Index out of range exceptions
- Elegant for-each loop
- **Python hopefully better error messages than C++**

Python and garbage collection



garbage.py

```
a = [2, 5, 3]
a = [7, 4]
```

a gets new value

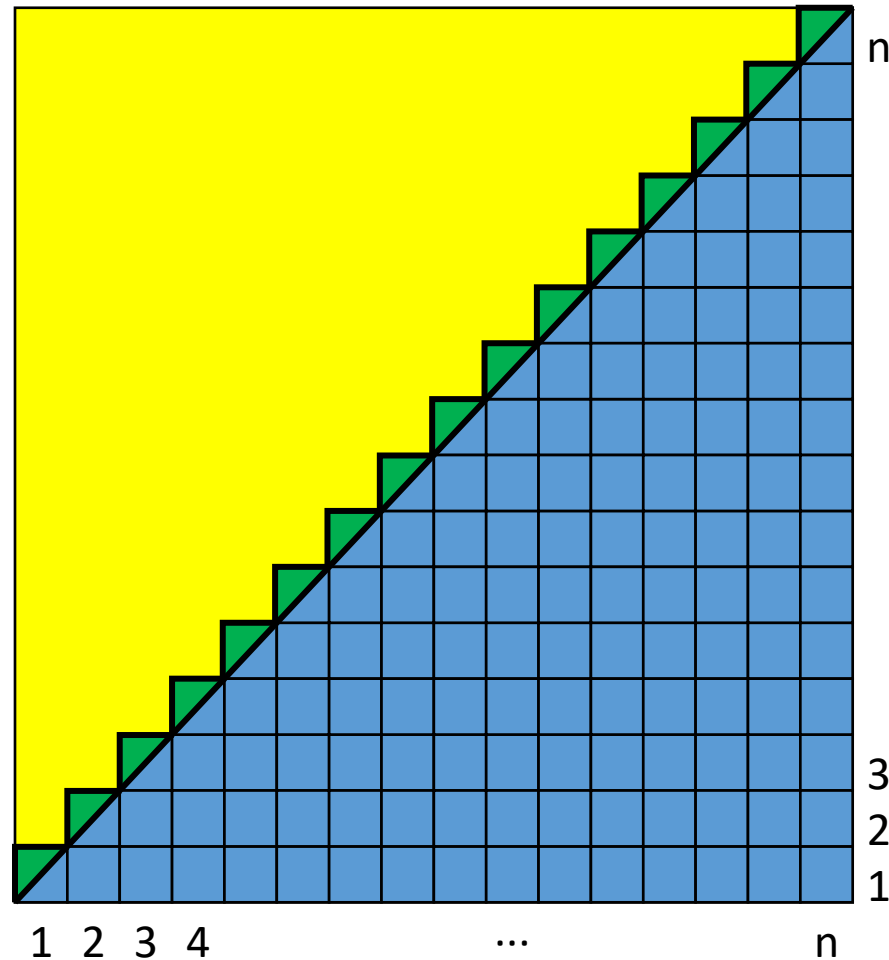
- Python and e.g. Java, C# and JavaScript have a **garbage collector** to automatically recycle garbage
- C and C++ garbage collection must be done explicitly by the program; forgetting to **free** memory again results in **memory leaks** – which can be really hard to find. **Have fun debugging!**
- Automatic garbage collection increases memory safety

Why Python ?

- Short concise code
- Index out of range exceptions
- Elegant for-each loop
- Python hopefully better error messages than C++
- **Garbage collection is done automatically**

Python performance vs C, C++ and Java

Compute sum $1 + 2 + 3 + \dots + n = n^2/2 + n/2$



$$1 + 2 + \dots + n$$

add.py

```
import sys

n = int(sys.argv[1])
sum = 0
for i in range(1, n + 1):
    sum += i
print("Sum =", sum)
```

add.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    int sum = 0;
    for (int i = 1; i <= n; i++)
        sum += i;
    printf("Sum = %d\n", sum);
}
```

add.cpp

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    int sum = 0;
    for (int i = 1; i <= n; i++)
        sum += i;
    cout << "Sum = " << sum << endl;
}
```

add.java

```
class Add{
    public static void main(String args[]){
        int n = Integer.parseInt(args[0]);
        int sum = 0;
        for (int i = 1; i <= n; i++)
            sum += i;
        System.out.println("Sum = " + sum);
    }
}
```

Timing results

Python							
n	C (gcc 9.2)	C++, int (g++ 9.2)	C++, long (g++ 9.2)	Java (12.0)	CPython (3.8.1)	PyPy (7.3.0)	Numba, int64
10 ⁷	0.001 sec*	0.001 sec*	0.003 sec	0.006 sec*	1.5 sec	0.27 sec	0.002 sec
10 ⁹	0.10 sec**	0.10 sec**	0.30 sec	0.40 sec**	145 sec	27 sec	0.2 sec

Wrong output (overflow)

* -2004260032 instead of 50000005000000

** -243309312 instead of 500000000500000000

- since C, C++, and Java only uses 32 bits to represent integers (and 64 bits for "long" integers)

Have fun
debugging!

```
Bit          6666666666555555555544444444443333333333222222222211111111110000000000
position      987654321098765432109876543210987654321098765432109876543210
bin(10**9)                                111011100110101100101000000000
bin(50000005000000)                      1011010111100110001000100010010110101101000000
bin(-2004260032+2**32)                    10001000100010010110101101000000
bin(500000000500000000)                  11011110000010110110101100111100010110010100000000
bin(-243309312+2**32)                    11110001011111110110010100000000
```

Timing results

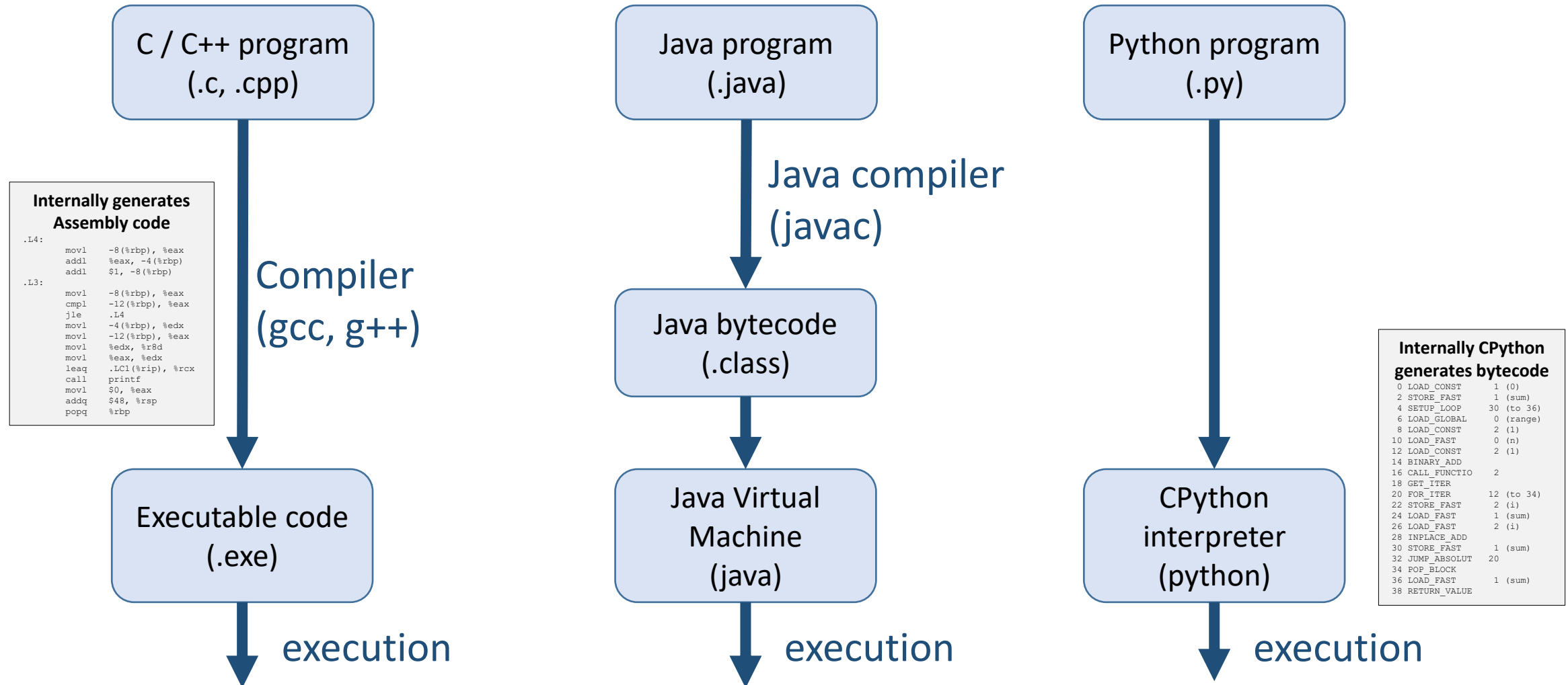
n	C (gcc 9.2)	C++, int (g++ 9.2)	C++, long (g++ 9.2)	Java (12.0)	Python		
					Python (3.8.1)	PyPy (7.3.0)	Numba, int64
10 ⁷	0.001 sec*	0.001 sec*	0.003 sec	0.006 sec*	1.5 sec	0.27 sec	0.002 sec
10 ⁹	0.10 sec**	0.10 sec**	0.30 sec	0.40 sec**	145 sec	27 sec	0.2 sec

- Relative speed

C ≈ C++ > Java >> Python

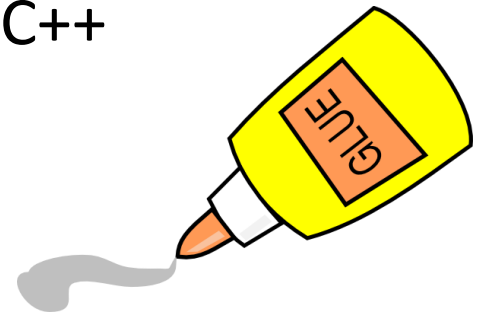
- C, C++, Java need to care about integer overflows – select integer representation carefully with sufficient number of bits (8, 16, 32, 64, 128)
- Python natively works with arbitrary long integers (as memory on your machine allows). Also possible in Java using the class `java.math.BigInteger`
- Python programs can (sometimes) run faster using PyPy
- Number crunching in **Python** should be delegated to **specialized modules (e.g. Numpy, CPLEX, Numba)** – often written in C or C++ and requires selecting right integer representation

Interpreter vs Compiler



Why Python ?

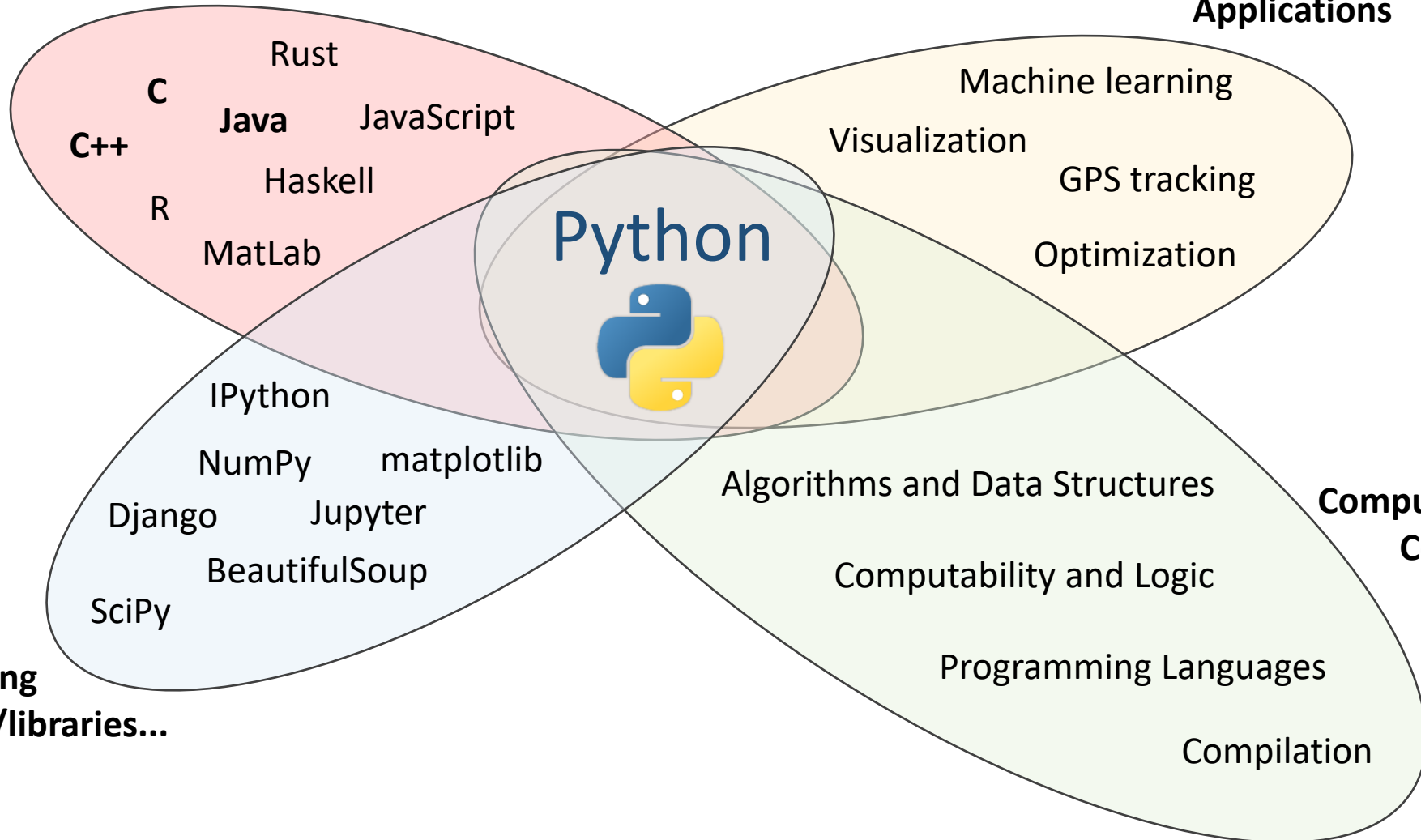
- Short concise code
- Index out of range exceptions
- Elegant for-each loop
- Python hopefully better error messages than C++
- Garbage collection is done automatically
- **Exact integer arithmetic (no overflows)**
- **Can delegate number crunching to C, C++, ...**



This course

Programming Languages

**(Scientific)
Applications**



**Programming
modules/packages/libraries...**

Course overview

Basic programming
Advanced / specific python
Libraries & applications

1. Introduction to Python	10. Functions as objects	19. Linear programming
2. Python basics / if	11. Object oriented programming	20. Generators, iterators, with
3. Basic operations	12. Class hierarchies	21. Modules and packages
4. Lists / while / for	13. Exceptions and files	22. Working with text
5. Tuples / comprehensions	14. Doc, testing, debugging	23. Relational data
6. Dictionaries and sets	15. Decorators	24. Clustering
7. Functions	16. Dynamic programming	25. Graphical user interfaces (GUI)
8. Recursion	17. Visualization and optimization	26. Java vs Python
9. Recursion and Iteration	18. Multi-dimensional data	27. Final lecture

10 handins
1 final project (last 1 month)

History of Python development

- Python created by Guido van Rossum in 1989, first release 0.9.0 1991
- Python 2 → Python 3 (clean up of Python 2 language)
 - Python 2 – version 2.0 released 2000, final version 2.7 released mid-2010
 - Python 3 – released 2008, current release 3.13.1
- Python 3 is *not* backward compatible, libraries incompatible

Python 2	Python 3
<code>print 42</code>	<code>print(42)</code>
<code>int</code> = C long (32 bits)	<code>int</code> = arbitrary number of digits (= named “long” in Python 2)
<code>7/3</code> → 2 returns “int”	<code>7/3</code> → 2.333... returns “float”
<code>range()</code> returns list (memory intensive)	<code>range()</code> returns iterator (memory efficient; xrange in Python 2)

Python.org

The screenshot shows the Python.org homepage with a dark blue header and navigation bar. The header includes links for Python, PSF, Docs, PyPI, Jobs, and Community. The main navigation bar has links for About, Downloads, Documentation, Community, Success Stories, News, and Events. A search bar with a 'GO' button and a 'Socialize' button are on the right. A red arrow points from the 'Docs' link to the 'Documentation' link in the main navigation bar. Another red arrow points from the 'PyPI' link to the search bar, with the text '+500.000 Python packages' next to it. A third red arrow points from the 'Downloads' link to the 'Download Python and IDLE' link in the main content area. The main content area features a code snippet on the left and an article titled 'All the Flow You'd Expect' on the right. The footer contains four columns: 'Get Started', 'Download', 'Docs', and 'Jobs', each with a brief description and a link.

Python

PSF

Docs

PyPI

Jobs

Community

python™

Donate

GO

Socialize

About Downloads Documentation Community Success Stories News Events

```
# For loop on a list
>>> numbers = [2, 4, 6, 8]
>>> product = 1
>>> for number in numbers:
...     product = product * number
...
>>> print('The product is:', product)
The product is: 384
```

>_

All the Flow You'd Expect

Python knows the usual control flow statements that other languages speak — `if`, `for`, `while` and `range` — with some of its own twists, of course. [More control flow tools in Python 3](#)

1 2 3 4 5

Python is a programming language that lets you work quickly and integrate systems more effectively. >>> [Learn More](#)

Get Started

Whether you're new to programming or an experienced developer, it's easy to learn and use Python.

[Start with our Beginner's Guide](#)

Download

Python source code and installers are available for download for all versions!

Latest: [Python 3.11.1](#)

Docs

Documentation for Python's standard library, along with tutorials and guides, are available online.

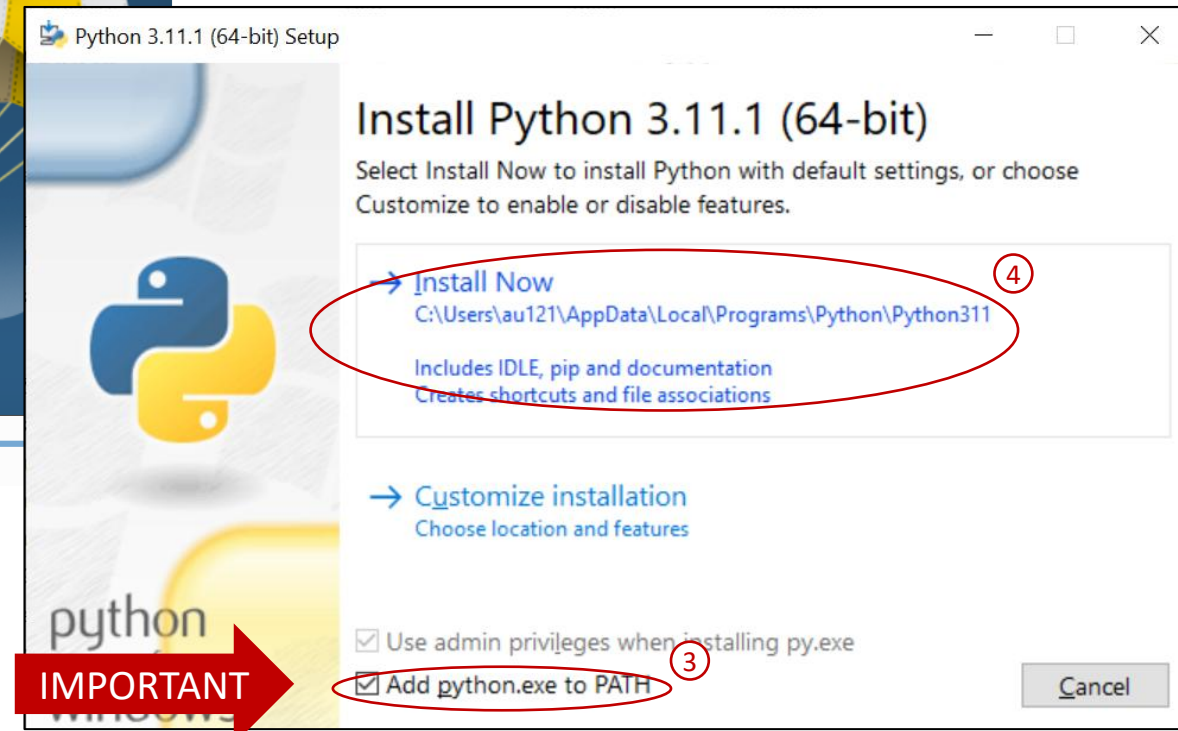
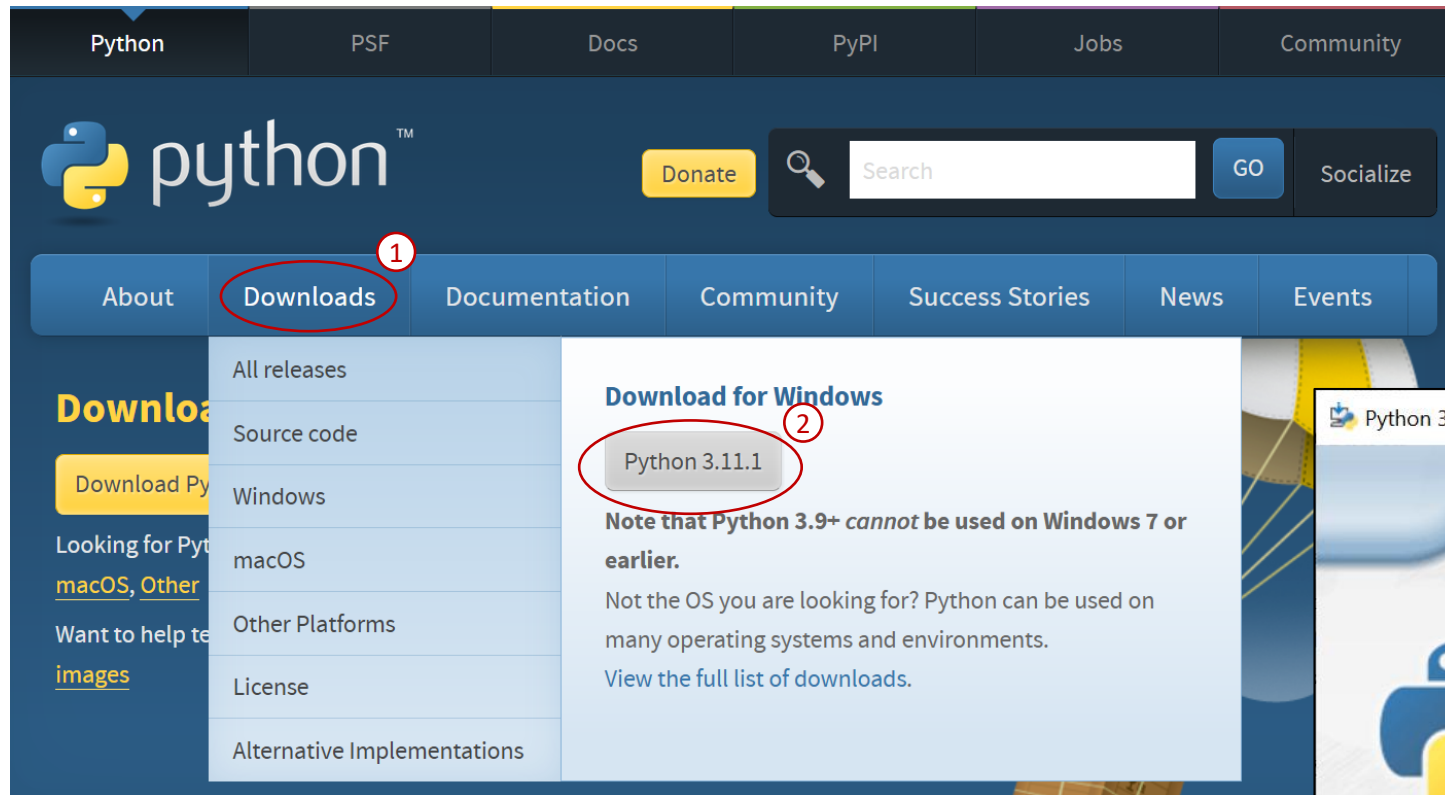
[docs.python.org](#)

Jobs

Looking for work or have a Python related position that you're trying to hire for? Our **relaunched community-run job board** is the place to go.

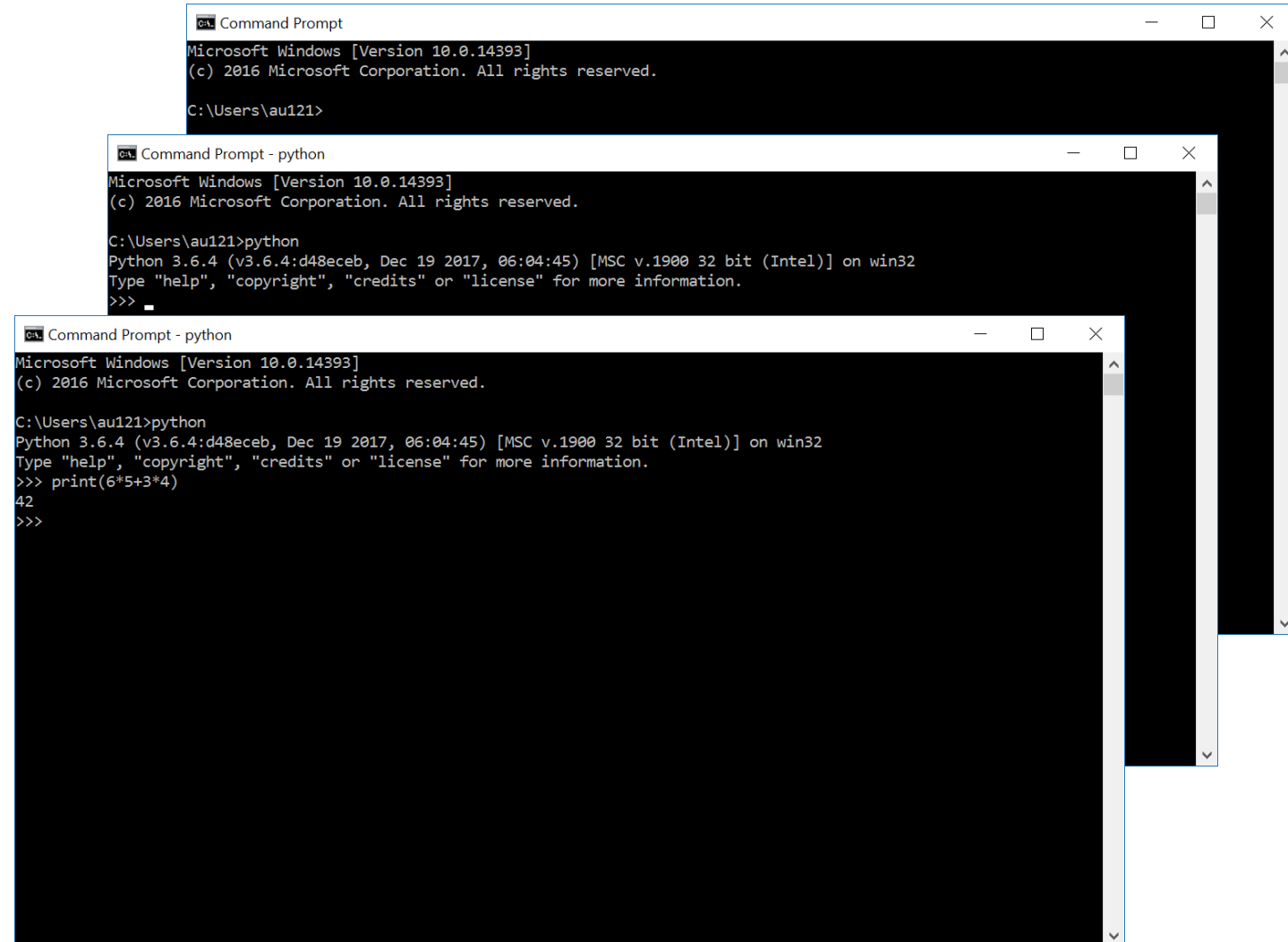
[jobs.python.org](#)

Installing Python



Running the Python Interpreter from a terminal

- Open Command Prompt (Windows-key + cmd)
- Type “python” + return
- Start executing Python statements
- To exit shell:
Ctrl-Z + return *or*
exit() + return
- Note: Sometimes “python” is installed as “python3”



The image displays three overlapping screenshots of a Windows Command Prompt window, illustrating the process of running the Python interpreter. The top window shows the standard Windows boot-up text and the user's prompt at C:\Users\au121>. The middle window shows the user typing 'python' and the system outputting the Python version (3.6.4) and the prompt changing to >>>. The bottom window shows the user typing a print statement 'print(6*5+3*4)' and the system outputting the result '42'.

```
Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>

Command Prompt - python
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

Command Prompt - python
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print(6*5+3*4)
42
>>>
```

Installing IPython –

A more powerful interactive Python shell

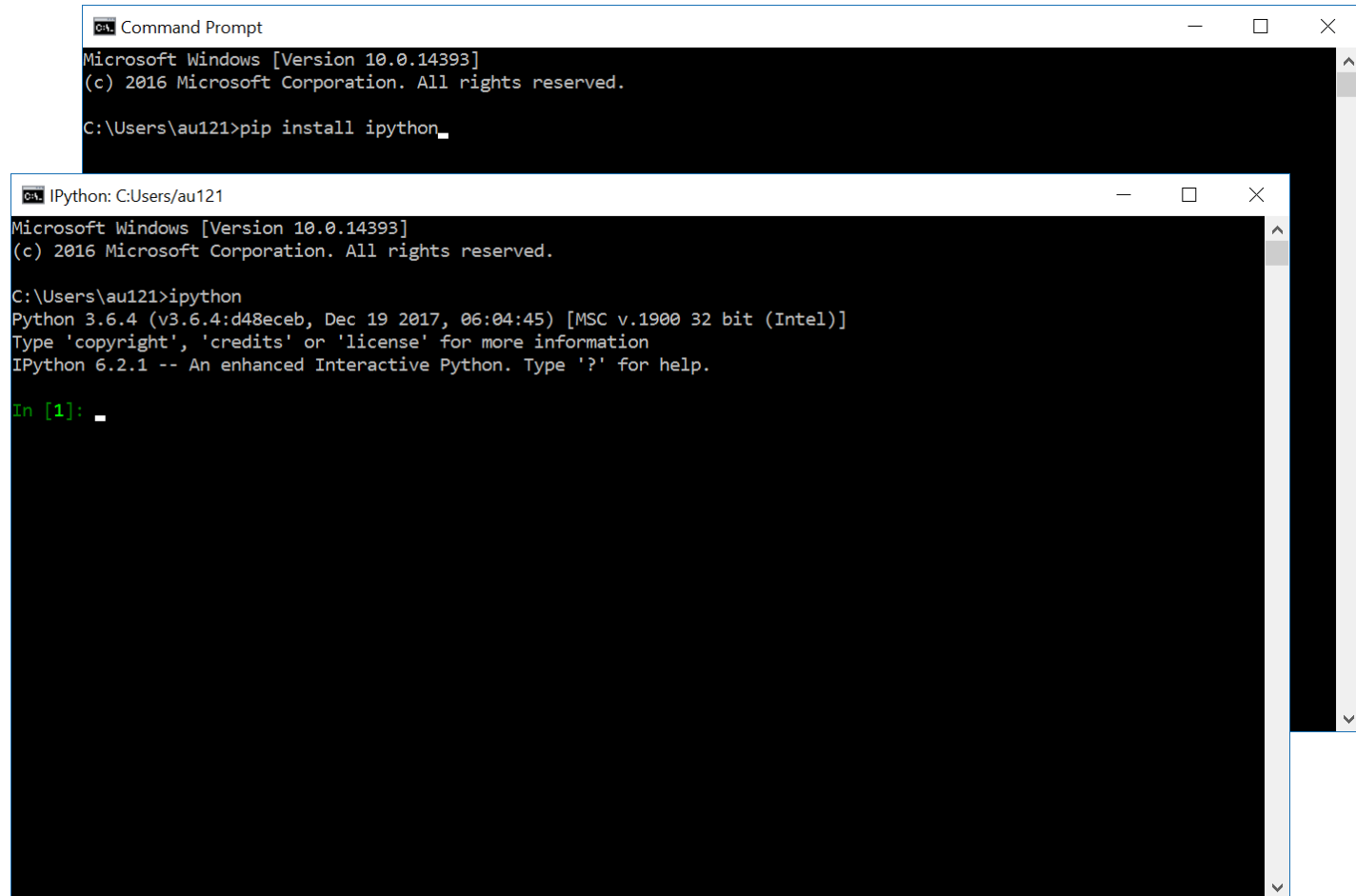
- Open Command Prompt
- Execute:

```
pip install ipython
```

- Start IPython

```
ipython
```

- pip = the Python package manager
- Note: Sometimes “pip” is installed as “pip3”



The screenshot shows two overlapping windows. The top window is titled 'Command Prompt' and displays the following text: 'Microsoft Windows [Version 10.0.14393] (c) 2016 Microsoft Corporation. All rights reserved. C:\Users\au121>pip install ipython_'. The bottom window is titled 'IPython: C:\Users\au121' and displays the following text: 'Microsoft Windows [Version 10.0.14393] (c) 2016 Microsoft Corporation. All rights reserved. C:\Users\au121>ipython Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] Type 'copyright', 'credits' or 'license' for more information IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help. In [1]: _'.

Some other usefull packages

- Try installing some more Python packages:

```
pip install numpy
```

linear algebra support (N-dimensional arrays)

```
pip install scipy
```

numerical integration and optimization

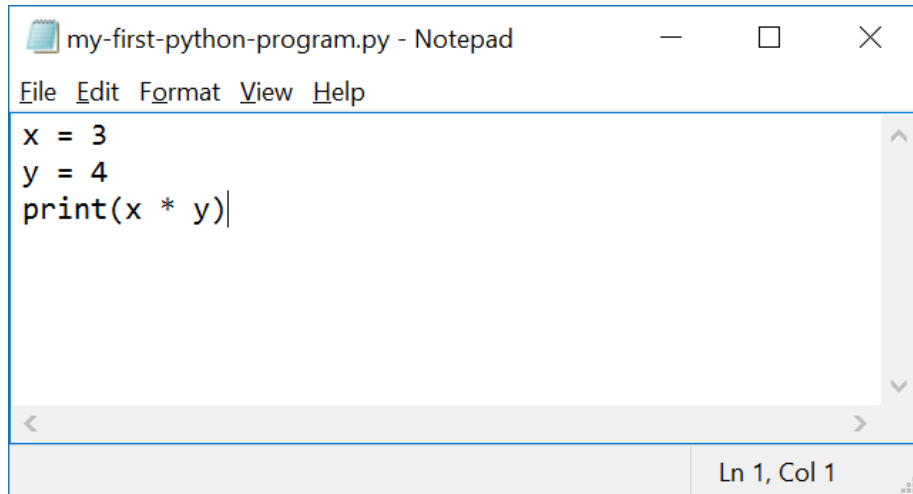
```
pip install matplotlib
```

2D and 3D plotting library

```
pip install pylint
```

Python source code analyzer enforcing a coding standard

Creating a Python program the very basic way

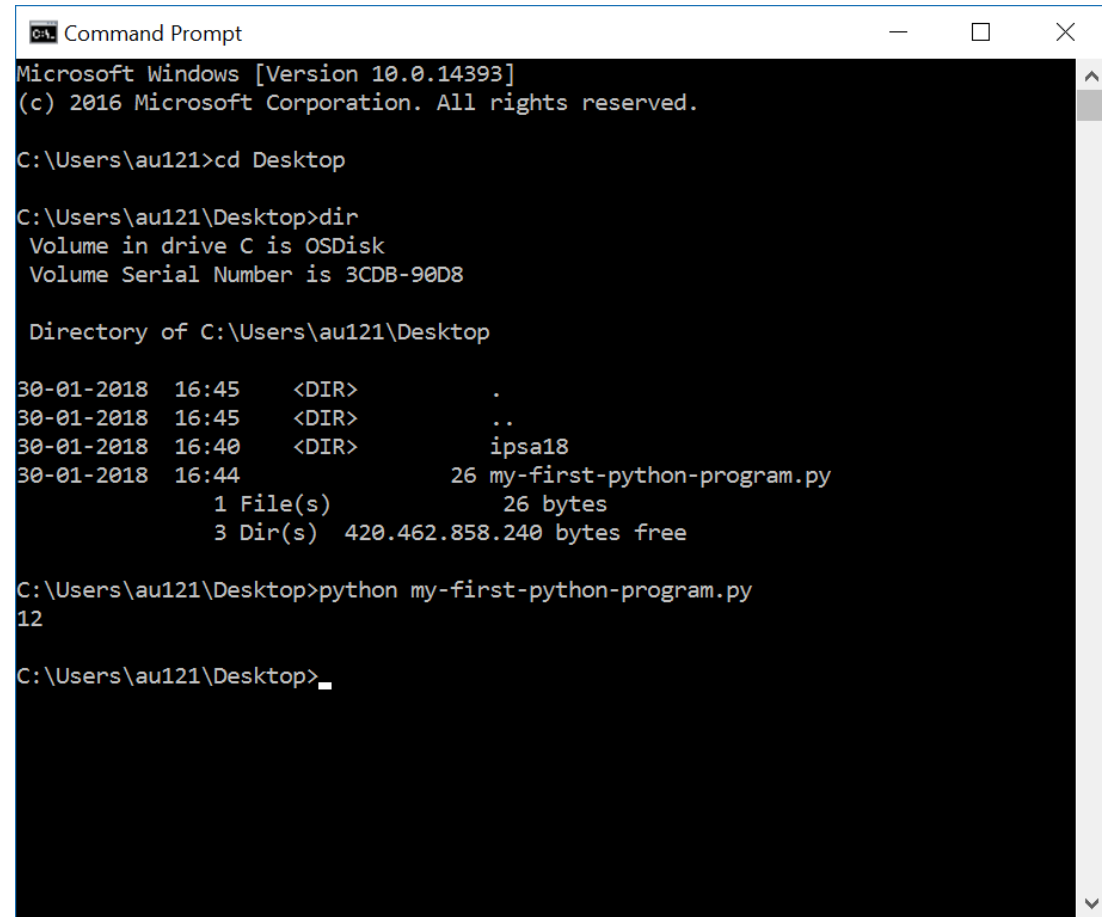
A screenshot of a Notepad window titled "my-first-python-program.py - Notepad". The window contains the following Python code:

```
x = 3
y = 4
print(x * y)
```

The status bar at the bottom right indicates "Ln 1, Col 1".

```
my-first-python-program.py - Notepad
File Edit Format View Help
x = 3
y = 4
print(x * y)
Ln 1, Col 1
```

- Open Notepad (or TextEdit on Mac)
 - write a simple Python program
 - save it
- Open a command prompt
 - go to folder (using cd)
 - run the program using
`python <program name>.py`

A screenshot of a Windows Command Prompt window. The user navigates to the Desktop directory and lists its contents, showing the file "my-first-python-program.py". Then, the user runs the program using the command "python my-first-python-program.py", which outputs the number "12".

```
Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>cd Desktop

C:\Users\au121\Desktop>dir
Volume in drive C is OSDisk
Volume Serial Number is 3CDB-90D8

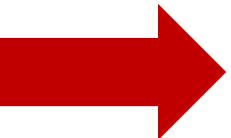
Directory of C:\Users\au121\Desktop

30-01-2018  16:45    <DIR>          .
30-01-2018  16:45    <DIR>          ..
30-01-2018  16:40    <DIR>          ipsa18
30-01-2018  16:44                26 my-first-python-program.py
               1 File(s)                26 bytes
               3 Dir(s)  420.462.858.240 bytes free

C:\Users\au121\Desktop>python my-first-python-program.py
12

C:\Users\au121\Desktop>
```

... or open IDLE and run program with F5



enable
line numbers
under options

```
my-first-python-program.py - C:\Users\au121\Desktop\my-first-python-program.py (3.11.0)
File Edit Format Run Options Window Help
1 x = 3
2 y = 4
3 print(x * y)
4 |
Ln: 4 Col: 0
```

```
IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\au121\Desktop\my-first-python-program.py =====
12
>>>
Ln: 6 Col: 0
```

- IDLE ships with Python from python.org
- Good beginner IDE (Integrated Development Environment)

The Python Ecosystem

- **Interpreters/compiler**

- CPython – reference C implementation from python.org
- PyPy – written in RPython (a subset of Python) – faster than Cpython
- Jython – written in Java and compiles to Java bytecode, runs on the JVM
- IronPython – written in C#, compiles to Microsoft's Common Language Runtime (CLR) bytecode
- Cython – project translating Python-ish code to C

- **Shells (IPython, IDLE, Jupyter)**

- **Libraries/modules/packages**

- pypi.python.org/pypi (PyPI - the Python Package Index, +500.000 packages)

- **IDEs (Integrated development environment)**

- IDLE comes with Python (docs.python.org/3/library/idle.html)
- Anaconda w. Spyder, IPython (www.anaconda.com/download)
- Canopy (enthought.com/product/canopy)
- Visual Studio Code (code.visualstudio.com)
- Python tools for Visual Studio (github.com/Microsoft/PTVS)
- PyCharm (www.jetbrains.com/pycharm/)
- Emacs (Python mode and ElPy mode)
- Notepad++

← Good beginner Python IDE

← *“Visual Studio Code is used by more than twice as many developers than its nearest alternative”, [Stack overflow survey 2024](#)*

Try to google “[best ide python](#)”

- **Python Style guide (PEP8)**

- pylint, pep8, flake8

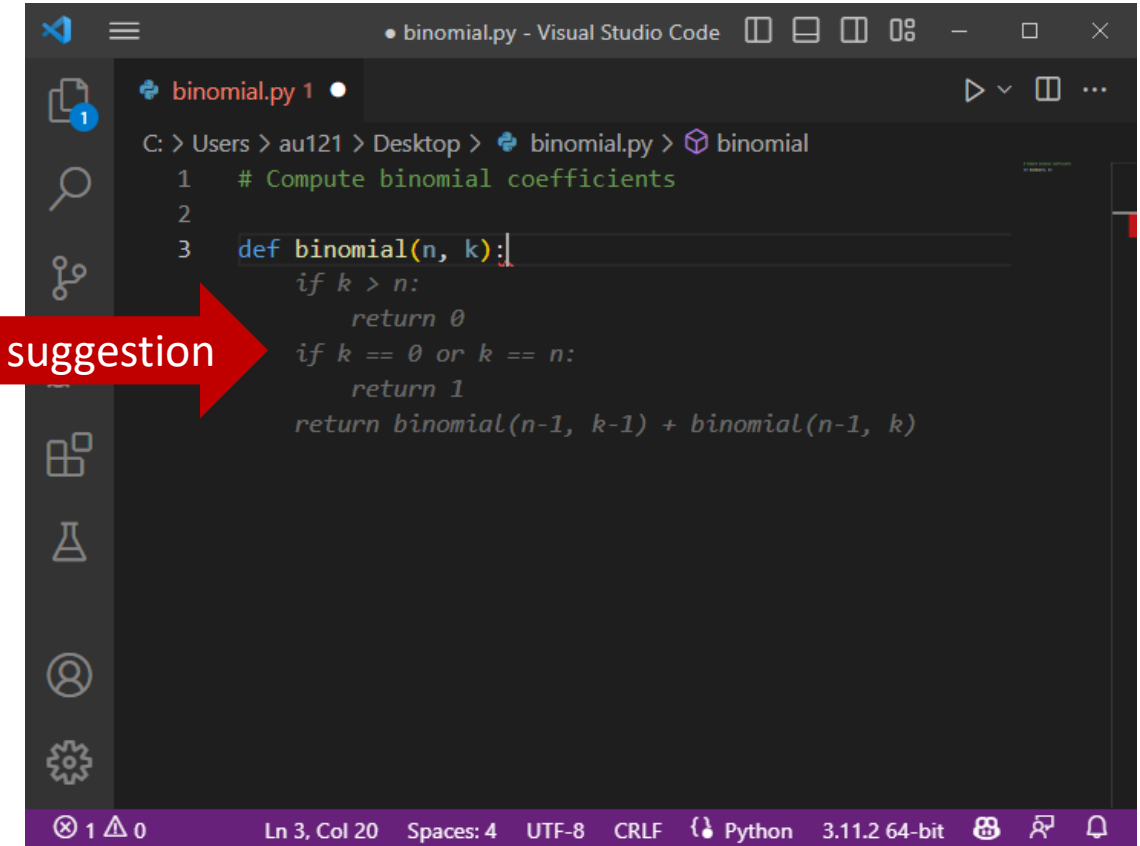
- **Python online**

- Google colab (colab.research.google.com), repl.it, sagemath.org, ...



IDEs and AI assistants

- Some IDEs integrate AI assistants to support code suggestions, e.g. GitHub Copilot in VS Code
- AI assistants increase productivity *if* you understand their output
- Interacting with an AI assistant can be a great programming tutor
-  AI assistants are not allowed at the exam



```
binomial.py 1
C: > Users > au121 > Desktop > binomial.py > binomial
1 # Compute binomial coefficients
2
3 def binomial(n, k):
    if k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binomial(n-1, k-1) + binomial(n-1, k)
```

Guido van Rossum, inventor of Python, on GitHub Copilot

"I use it every day. It writes a lot of code for me... and usually it is slightly wrong but it still saves me typing."

Python and the Future of Programming, Guido van Rossum interviewed by Lex Fridman

Python basics

- Comments
- `"""`
- Variable names
- `int`, `float`, `str`
- type conversion
- assignment (`=`)
- `print()`, `help()`, `type()`

Python comments

A '#' indicates the beginning of a comment.
From '#' until end of line is ignored by Python.

```
x = 42    # and here goes the comment
```

Comments useful to describe what a piece of code is supposed to do, what kind of input is expected, what is the output, side effects...

Comments are aimed at people (including yourself) reading the code

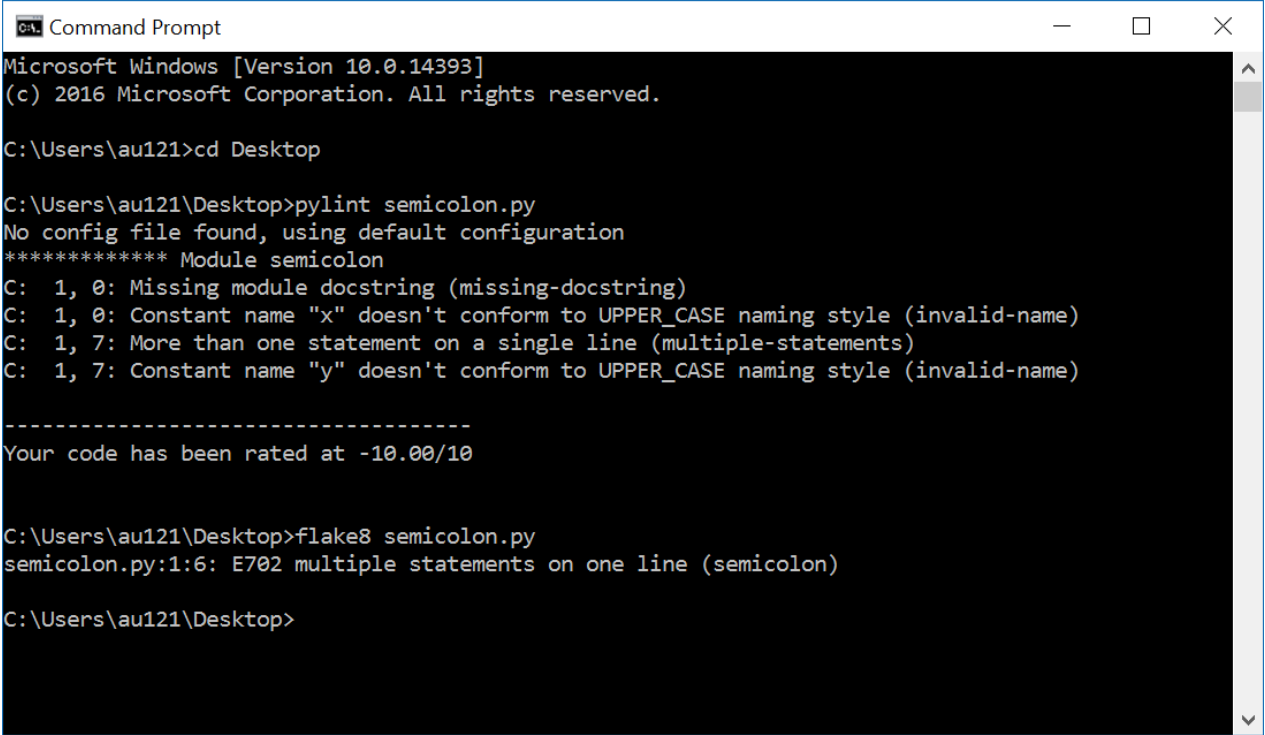
The “;” in Python

- Normally statements follow in consecutive lines with identical indentation

```
x = 1  
y = 1
```

- but Python also allows multiple statements on one line, separated by “;”

```
x = 1; y = 1
```



```
Command Prompt  
Microsoft Windows [Version 10.0.14393]  
(c) 2016 Microsoft Corporation. All rights reserved.  
  
C:\Users\au121>cd Desktop  
  
C:\Users\au121\Desktop>pylint semicolon.py  
No config file found, using default configuration  
***** Module semicolon  
C: 1, 0: Missing module docstring (missing-docstring)  
C: 1, 0: Constant name "x" doesn't conform to UPPER_CASE naming style (invalid-name)  
C: 1, 7: More than one statement on a single line (multiple-statements)  
C: 1, 7: Constant name "y" doesn't conform to UPPER_CASE naming style (invalid-name)  
  
-----  
Your code has been rated at -10.00/10  
  
C:\Users\au121\Desktop>flake8 semicolon.py  
semicolon.py:1:6: E702 multiple statements on one line (semicolon)  
  
C:\Users\au121\Desktop>
```

neither **pylint** or **flake8** like “;”

- General Python PEP 8 guideline: **avoid using “;”**
- Other languages like C, C++ and Java require “;” to end/separate statements

Variable names

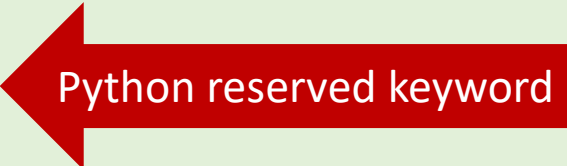
- Variable name = sequence of **letters** 'a'-'z', 'A'-'Z', **digits** '0'-'9', and **underscore** '_'

v, volume, height_of_box, WidthOfBox, x0, _v12_34B, _
(snake_case) (CamelCase)

- a name cannot start with a digit
 - names are case sensitive (AB, Ab, aB and ab are different variables)
- Variable names are **references to objects in memory**
 - **Use meaningful variables names**
 - **Python 3 reserved keywords:**

and	class	elif	for	import	nonlocal	raise	with	None
as	continue	else	from	in	not	return	yield	False
assert	def	except	global	is	or	try		True
break	del	finally	if	lambda	pass	while		

Question – Not a valid Python variable name?

- a) `print`
-  b) `for`  Python reserved keyword
- c) `_100`
- d) `x`
- e) `_`
- f) `python_for_ever`
- g) Don't know

Python shell

```
> print = 7
> print(42)
| Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|   TypeError: 'int' object is not callable
```



`print` is a valid variable name, with default value a builtin *function* to print output to a shell – assigning a new value to `print` is very likely a bad idea (like many others `sum`, `int`, `str`, ...)

Integer literals

- -4, -3, -2, -1, 0, 1, 2, 3, 4
- Python integers can have an arbitrary number of digits (only limited by machine memory)
- Can be preceded by a plus (+) or minus (-)
- Leading zeros are not allowed (e.g., 7 cannot be written as 007)
- For readability underscores (_) can be added between digits,

2_147_483_647

(for more, see [PEP 515 - Underscores in Numeric Literals](#))

Question – What statement will not fail?

a) $x = _42$



b) $_10 = -1_1$

c) $x = 1___0$

d) $x = +1_0_$

e) Don't know

Float literals

- Decimal numbers are represented using **float** – contain “.” or “e”
- Examples
 - 3.1415
 - -.00134
 - $124e3 = 124 \cdot 10^3$
 - $-2.345e2 = -234.5$
 - $12.3e-4 = 0.00123$
-  Floats are often only approximations, e.g. 0.1 is *not* 1/10
- Extreme values (CPython)
 - `max = 1.7976931348623157e+308`
 - `min = 2.2250738585072014e-308`
- NB: Use module `fractions` for exact fractions/rational numbers.



Python shell

```
> 0.1 + 0.2 + 0.3
| 0.60000000000000001
> (0.1 + 0.2) + 0.3
| 0.60000000000000001
> 0.1 + (0.2 + 0.3)
| 0.6
> type(0.1)
| <class 'float'>
> 1e200 * 1e300
| inf
> 0.1+(0.2+0.3) == (0.1+0.2)+0.3
| False
> x = 0.1 + 0.2
> y = 0.3
> x == y
| False
> print(f'{x:.30f}') # 30 decimals
| 0.30000000000000000044408920985006
> print(f'{y:.30f}') # 30 decimals
| 0.2999999999999999988897769753748
> import sys
> sys.float_info.min
| 2.2250738585072014e-308
> sys.float_info.max
| 1.7976931348623157e+308
```

Associativity rule does
not apply to floats

Question – What addition order is “best”?



- a) $1e10 + 1e-10 + -5e-12 + -1e10$
- b) $1e10 + -1e10 + 1e-10 + -5e-12$
- c) $1e-10 + 1e10 + -1e10 + -5e-12$
- d) $-5e-12 + -1e10 + 1e10 + 1e-10$
- e) Any order is equally good
- f) Don't know

$1e10$	=	100000000000
$-1e10$	=	-100000000000
$1e-10$	=	0.00000000001
$-5e-12$	=	-0.000000000005

Python shell

```
> 1e10 + 1e-10 + -5e-12 + -1e10
| 0.0
> 1e10 + -1e10 + 1e-10 + -5e-12
| 9.50000000000000001e-11
> 1e-10 + 1e10 + -1e10 + -5e-12
| -5e-12
> -5e-12 + -1e10 + 1e10 + 1e-10
| 1e-10
```



a) - d) give four different outputs

Approximating $\pi = 3.14159265359\dots$

$$\frac{\pi^2}{6} = \sum_{k=1}^{+\infty} \frac{1}{k^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \dots$$
$$= 1.6449340668\dots$$

Riemann zeta function $\zeta(2)$

`pi_approximation_riemann.py`

```
apx = 0.0
k = 0.0
while True:
    k = k + 1.0
    apx = apx + 1.0 / (k * k)
    print(k, apx)
```

Output

```
...
94906261.0 1.6449340578345741
94906262.0 1.6449340578345744
94906263.0 1.6449340578345746
94906264.0 1.6449340578345748
94906265.0 1.644934057834575
94906266.0 1.644934057834575
94906267.0 1.644934057834575
94906268.0 1.644934057834575
94906269.0 1.644934057834575
94906270.0 1.644934057834575
...
```



This is not a course in numeric computations – but now you are warned....

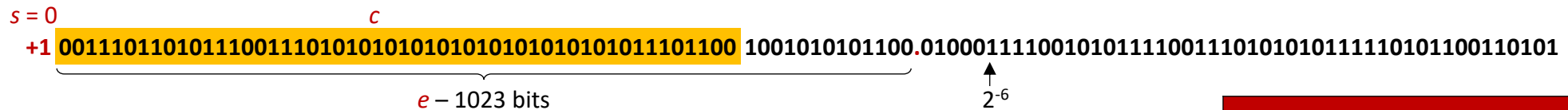
Python float $\equiv$ IEEE-754 double precision*

- A binary number is a number in base 2 with digits/bits from $\{0,1\}$

$$10110_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 4 + 2 = 22_{10}$$

- IEEE-754 64-bit double

sign s	exponent e	coefficient c
1 bit	11 bits	52 bits



Float value	Case
$(-1)^s \cdot (1 + c \cdot 2^{-52}) \cdot 2^{e-1023}$	$0 < e < 2047$
$(-1)^s \cdot c \cdot 2^{-1074}$	$e = 0, c \neq 0$
+0 and -0	$e = 0, c = 0$
$+\infty$ and $-\infty$	$e = 2047, c = 0$
NaN (“not a number”)	$s = 0, e = 2047, c \neq 0$

Python shell

```
> 1e200 * 1e200
| inf
> -1e200 * 1e200
| -inf
> 1e-200 * 1e-200
| 0.0
> -1e-200 * 1e-200
| -0.0
> 1e200 * 1e200 * 0.0
| nan
```

(*most often, but there is no guarantee given in the Python language specification that floats are represented using IEEE-754)

String literals (type `str`)

- Sequence of characters enclosed by single (') or double (") quotes

```
"a 'quoted' word"      "Hello World"      'abc'
'a "quoted" word'      '_ _ \' _ _'
```

- Escape characters

<code>\n</code>	newline
<code>\t</code>	tab
<code>\\</code>	backslash
<code>\'</code>	single quote
<code>\"</code>	double quote
<code>\N{</code>	<i>unicode name</i> }

- A backslash (\) at the end of line, will continue line/string on next line
- Use triple single or double quotes (' ' ' or " " ") for enclosing strings spanning more lines
(in particular for Python Docstrings, see [PEP 257](#))

string-test.py

```
print("abc")
print('de\'f')
print("'ghi'")
print("'jk\nl'\\"")
print("mn\
o")
print("p\\q\\tr")
print("\\N{SNOWMAN}")
```

Output

```
$ python string-test.py
abc
de'f
'ghi'
'jk
l'"
mno
p\q      r
```



Question – What does the following print ?

```
print ("\\\\" "\n\n")
```

- a) \\\\" "\n\n"
- b) \"\nn"
-  c) \"\n
'
- d) "nn"
- e) \"
'
- f) Don't know

Long string literals

- Long string literals often need to be split over multiple lines
- In Python two (or more) **string literals following each other** will be treated as a single string literal (they can use different quotes)
- Putting **parenthesis** around multiple literals allows line breaks
- Advantages:
 - avoids the backslash at the end of line
 - can use indentation to increase readability
 - allows comments between literals

long-string-literals.py

```
s1 = 'abc' "def" # two string literals
print(s1)
s2 = ''' ''' ''' # avoid escaping quotes
print(s2)
s3 = 'this is a really, really, really, \
really, really, long string'
print(s3)
s4 = ('this is a really, really, '
      'really, really, really, '
      'long string')
print(s4)
very_very_long_variable_name = (
    'this is a really, really, ' # line 1
    'really, really, really, '   # line 2
    "long string"               # line 3
)
print(very_very_long_variable_name)
```

Python shell

```
| abcdef
| '''
| this is a really, really, really, really,
| really, long string
| this is a really, really, really, really,
| really, long string
| this is a really, really, really, really,
| really, long string
```


Raw string literals

- By prefixing a string literal with an `r`, the string literal will be considered a **raw string** and backslashes become literal characters
- Useful in cases where you actually need backslashes in your strings, e.g. when working with Python's regular expression module `re`

Python shell

```
> print('\let\epsilon\varepsilon')      # \v = vertical tab
| \let\epsilon
|  arepsilon
> print('\\let\\epsilon\\varepsilon')  # many backslashes
| \let\epsilon\varepsilon
> print(r'\let\epsilon\varepsilon')    # more readable
| \let\epsilon\varepsilon
```

- Strange exception: A raw string cannot end with an odd number of backslashes

print(...)

- `print` can print zero, one, or more values
- default behavior
 - print a space between values
 - print a line break after printing all values
- default behavior can be changed by **keyword arguments** “`sep`” and “`end`”

Python shell

```
> print()  
|  
> print(7)  
| 7  
> print(2, 'Hello')  
| 2 Hello  
> print(3, 'a', 4)  
| 3 a 4  
> print(3, 'a', 4, sep=':')  
| 3:a:4  
> print(5); print(6)  
| 5  
| 6  
> print(5, end=', '); print(6)  
| 5, 6
```

print(...) and help(...)

Python shell

```
> help(print)
| Help on built-in function print in module builtins:
|
| print(...)
|     print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
|
|     Prints the values to a stream, or to sys.stdout by default.
|     Optional keyword arguments:
|     file: a file-like object (stream); defaults to the current sys.stdout.
|     sep:  string inserted between values, default a space.
|     end:  string appended after the last value, default a newline.
|     flush: whether to forcibly flush the stream.
```

Assignments

- *variable = expression*

`x = 42`

- Multiple assignments – right hand side evaluated before assignment

`x, y, z = 2, 5, 7`

- Useful for swapping

`x, y = y, x`

- Assigning multiple variables same value in left-to-right

`x = y = z = 7`



Warning

`i = 1`

`i = v[i] = 3` # `v[3]` is assigned value 3

In languages like C and C++ instead
`v[1]` is assigned 3

Python is dynamically typed, type(...)

- The current type of a value can be inspected using the **type()** function (that returns a type object)
- In Python the values contained in a variable over time can be of different type
- In languages like C, C++ and Java variables are declared with a given type, e.g.

```
int x = 42;
```

and the different values stored in this variable must remain of this type

Python shell

```
> x = 1
> type(x)
| <class 'int'>
> x = 'Hello'
> type(x)
| <class 'str'>
> type(42)
| <class 'int'>
> type(type(42))
| <class 'type'>
```



x new type

Type conversion

- Convert a value to another type:

new-type(value)

- Sometimes done automatically:

`1.0+7=1.0+float(7)=8.0`

Python shell

```
> float(42)
| 42.0
> int(7.8)
| 7
> x = 7
> print("x = " + x)
| Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
| TypeError: must be str, not int
> print("x = " + str(x))
| x = 7
> print("x = " + str(float(x)))
| x = 7.0
> int("7.3")
| Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
| ValueError: invalid literal for int() with base 10: '7.3'
> int(float("7.3"))
| 7
```



Questions – `str(float(int(float("7.5"))))` ?

a) 7

b) 7.0

c) 7.5

d) "7"

 e) "7.0"

f) "7.5"

g) Don't know

Control structures

- `input()`
- `if-elif-else`
- `while-break-continue`

input

- The builtin function `input(message)` prints *message*, and waits for the user provides a line of input and presses return. The line of input is returned as a `str`
- If you e.g. expect input to be an `int`, then remember to convert the input using `int()`

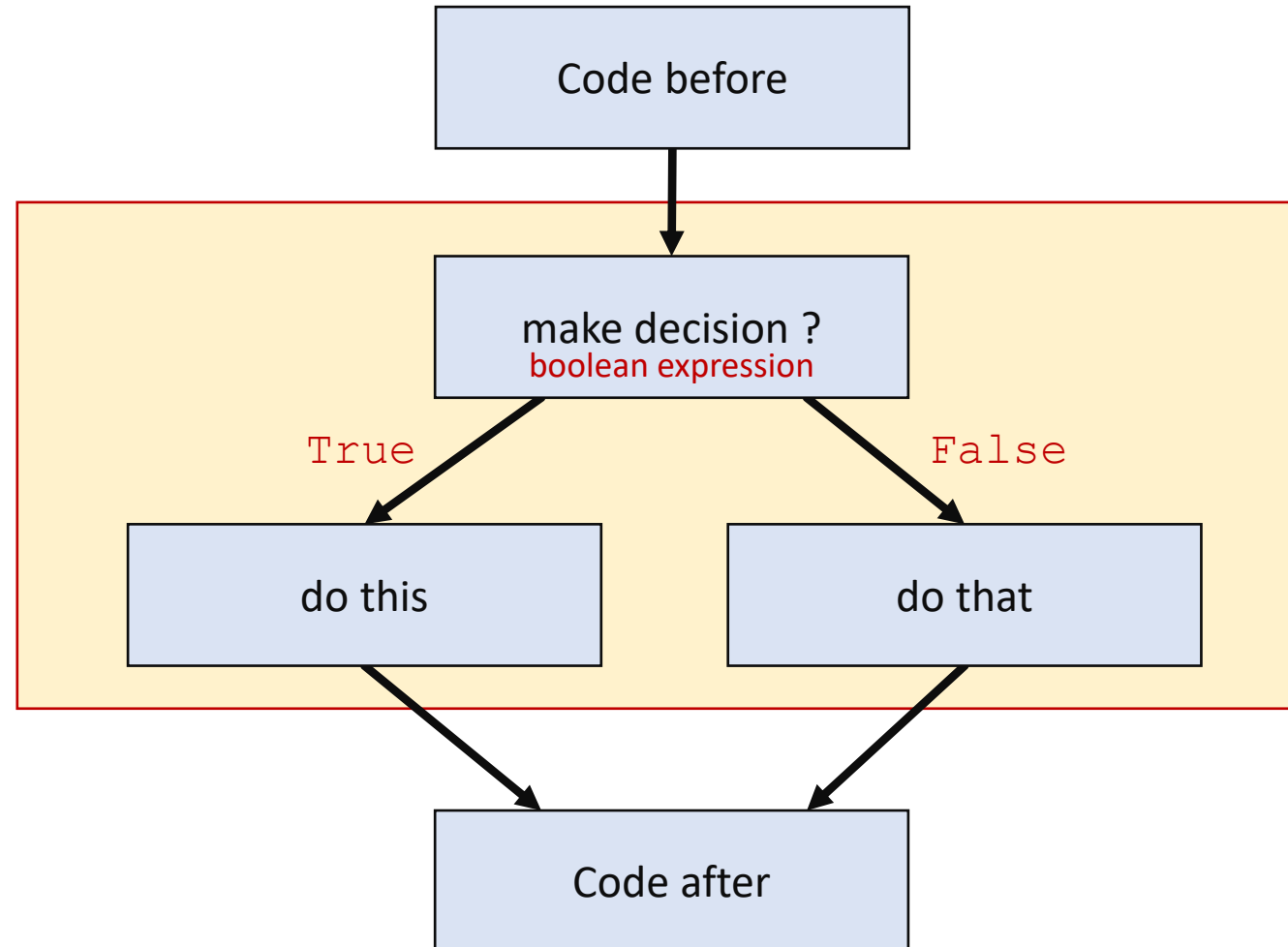
name-age.py

```
name = input('Name: ')
age = int(input('Age: '))
print(name, 'is', age, 'years old')
```

Python shell

```
> Name: Donald Duck
> Age: 84
| Donald Duck is 84 years old
```

Branching – do either this or that ?



Basic if-else

`if` *boolean expression*:

identical
indentation { *code*
code
code

`else:`

identical
indentation { *code*
code
code

if-else.py

```
if x % 2 == 0:
    print('even')
else:
    print('odd')
```

Identical indentation for a sequence of lines = the same spaces/tabs should precede code

pass

- `pass` is a Python statement doing nothing. Can be used where a statement is required but you want to skip (e.g. code will be written later)
- Example (bad example, since `else` could just be omitted):

if-else.py

```
if x % 2 == 0:  
    print('even')  
else:  
    pass
```

if-elif-else

`if condition:`

`code`

`elif condition:` # zero or more “elfi” $\equiv$ “else if”

`code`

`else:` # optional

`code`

```
if (condition) {  
    code  
} else if (condition) {  
    code  
} else {  
    code  
}
```

Java, C, C++ syntax

Other languages using indentation for blocking:
ABC (1976), occam (1983), Miranda (1985)

if.py

```
if x == 0:  
    print('zero')
```

if-else.py

```
if x % 2 == 0:  
    print('even')  
else:  
    print('odd')
```

elif.py

```
if x < 0:  
    print('negative')  
elif x == 0:  
    print('zero')  
elif x == 1:  
    print('one')  
else:  
    print('>= 2')
```

elif can make code nicer (less indentation)

elif.py

```
if x < 0:
    print('negative')
elif x == 0:
    print('zero')
elif x == 1:
    print('one')
else:
    print('>= 2')
```

ugly-if.py

```
if x < 0:
    print('negative')
else:
    if x == 0:
        print('zero')
    else:
        if x == 1:
            print('one')
        else:
            print('>= 2')
```

Questions – What value is printed?

```
x = 1
if x == 2:
    x = x + 1
else:
    x = x + 1
    x = x + 1
x = x + 1
print(x)
```

a) 1

b) 2

c) 3



d) 4

e) 5

f) Don't know

Nested if-statements

nested-if.py

```
if x < 0:
    print('negative')
elif x % 2 == 0:
    if x == 0:
        print('zero')
    elif x == 2:
        print('even prime number')
    else:
        print('even composite number')
else:
    if x == 1:
        print('one')
    else:
        print('some odd number')
```


Common mistake

if-if.py

```
x = int(input())  
if x == 0:  
    print('zero')  
if x % 2 == 0:  
    print('even')
```

Python shell

```
> 0  
| zero  
| even
```



if-elif.py

```
x = int(input())  
if x == 0:  
    print('zero')  
elif x % 2 == 0:  
    print('even')
```

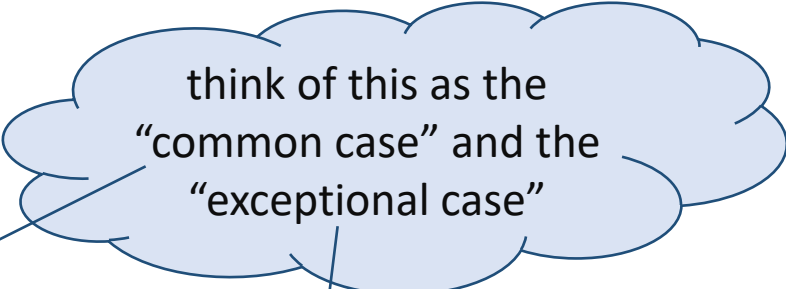
Python shell

```
> 0  
| zero
```

if-else *expressions*

- A very common computation is

```
if test:  
    x = true-expression  
else:  
    x = false-expression
```



think of this as the
“common case” and the
“exceptional case”

- In Python there is a shorthand for this:

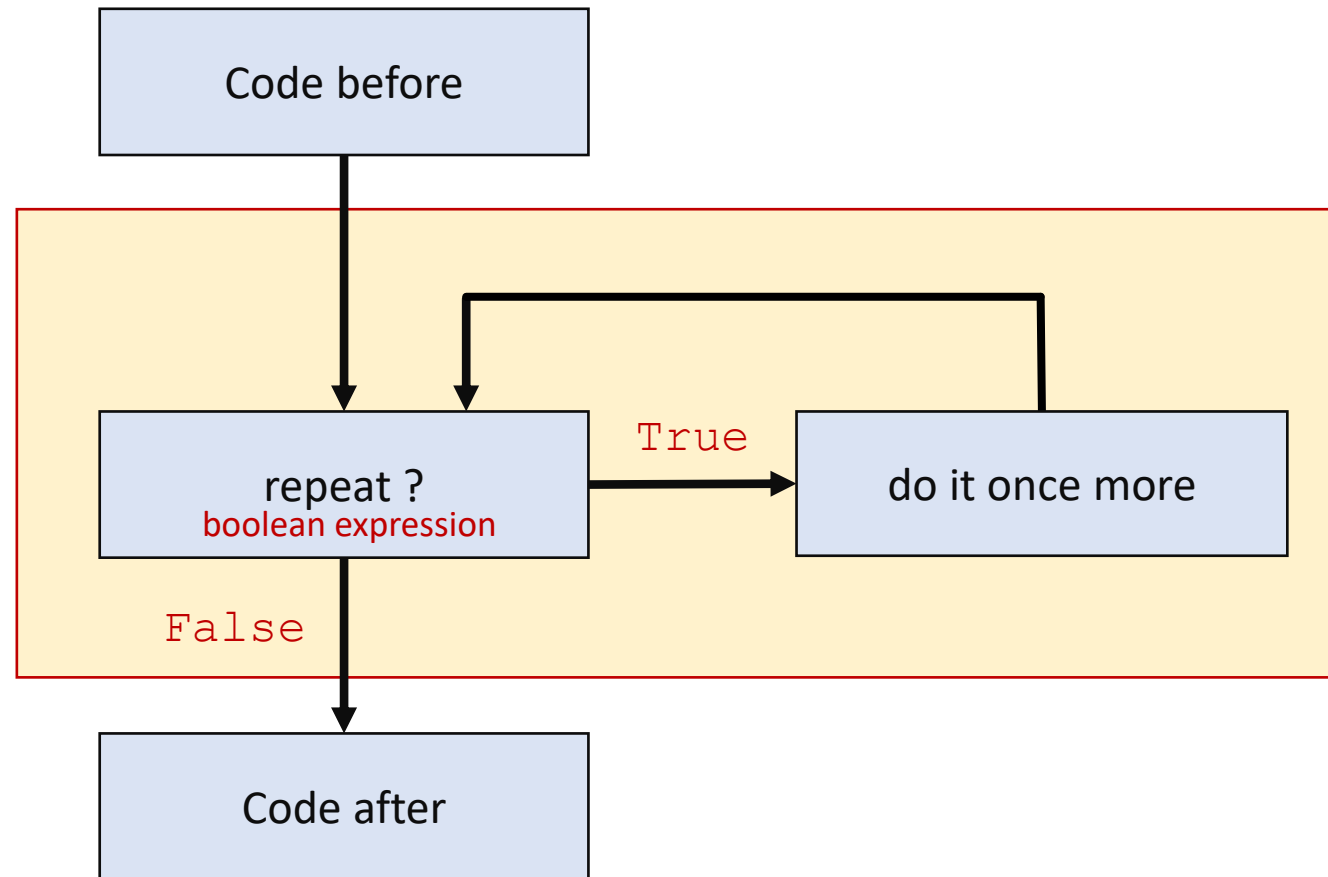
```
x = true-expression if test else false-expression
```

(see [What's New in Python 2.5 - PEP 308: Conditional Expressions](#))

- In C, C++, Java, Javascript the equivalent notation is (note the different order)

```
x = test ? true-expression : false-expression
```

Repeat until done



while-statement

`while condition:`

`code`

`...`

`break` # jump to code after while loop

`...`

`continue` # jump to condition at the

`...` # beginning of while loop

```
while (condition) {  
    code  
}
```

Java, C, C++ syntax

count.py

```
x = 1  
while x <= 5:  
    print(x, end=' ')  
    x = x + 1  
print('and', x)
```

Python shell

```
| 1 2 3 4 5 and 6
```

The function `randint(a, b)` from module `random` returns a random integer from $\{a, a + 1, \dots, b - 1, b\}$

random-pair.py

```
from random import randint  
while True:  
    x = randint(1, 10)  
    y = randint(1, 10)  
    if abs(x - y) >= 2:  
        break  
    print('too close', x, y)  
print(x, y)
```

Python shell

```
| too close 4 4  
| too close 10 9  
| 8 5
```

An exercise asks to
simplify the code

Computing $\lfloor \sqrt{x} \rfloor$ using binary search

int-sqrt.py

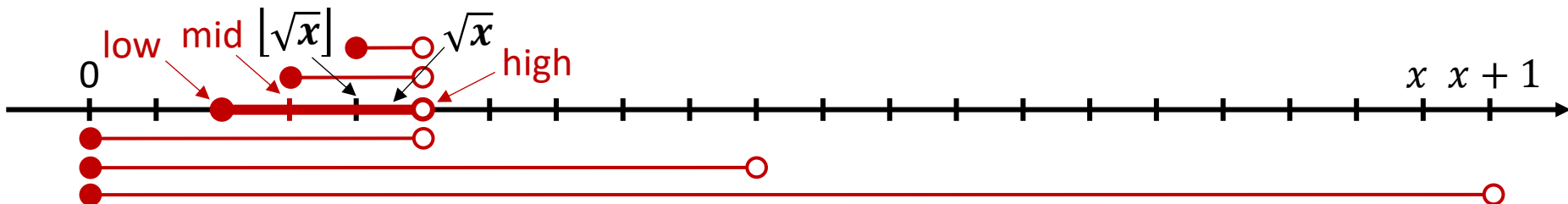
```
x = 20
low = 0
high = x + 1
while True: # low <= sqrt(x) < high
    if low + 1 == high:
        break
    mid = (high + low) // 2
    if mid * mid <= x:
        low = mid
        continue
    high = mid
print(low) # low = floor(sqrt(x))
```

Integer division

$$\left\lfloor \frac{\text{high} + \text{low}}{2} \right\rfloor$$

$$\text{mid} \leq \sqrt{x}$$

$$\Updownarrow$$
$$\text{mid}^2 \leq x$$



bisect

- **Note** Binary search on sorted lists is supported by the standard library module `bisect`
- `bisect_left` and `bisect_right` return the *insertion point* before and after, respectively, of existing occurrences of the value

binary_search_bisect.py

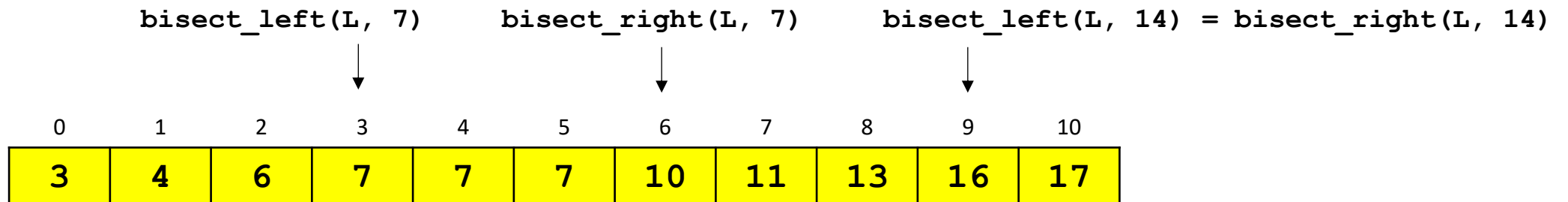
```
from bisect import bisect_left, bisect_right

L = [3, 4, 6, 7, 7, 7, 10, 11, 13, 16, 17]

print(bisect_left(L, 7))
print(bisect_right(L, 7))
print(bisect_left(L, 14))
print(bisect_right(L, 14))
```

Python shell

```
| 3 # bisect_left(L, 7)
| 6 # bisect_right(L, 7)
| 9 # bisect_left(L, 14)
| 9 # bisect_right(L, 14)
```

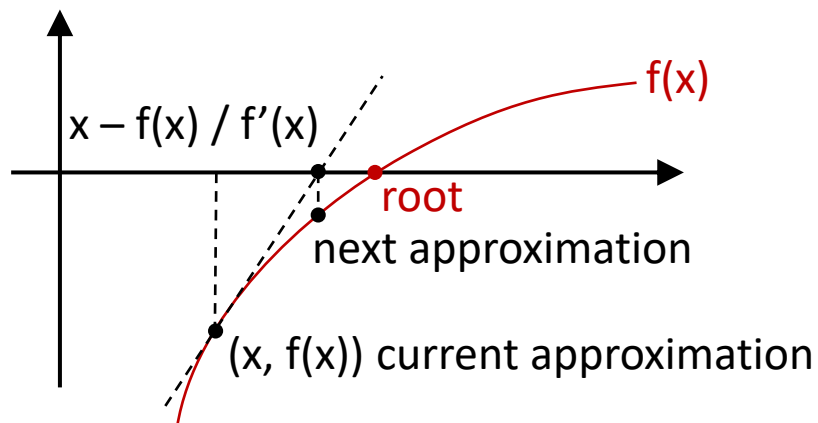


Division using the Newton-Raphson method

- **Goal:** Compute $1 / n$ only using +, -, and *
- $x = 1 / n \iff f(x) = n - 1 / x = 0$
- Problem reduces to finding **root** of f
- Newton-Raphson:

$$x := x - f(x)/f'(x) = x - (n - 1/x)/(1/x^2) = (2 - n \cdot x) \cdot x$$

since $f'(x) = 1 / x^2$ for $f(x) = n - 1 / x$



division.py

```
n = 0.75 # n in [0.5, 1.0]
x = 1.0
last = 0.0
while last < x:
    print(x)
    last = x
    x = (2 - n * x) * x
print('Apx of 1.0 /', n, '=', x)
print('Python 1.0 /', n, '=', 1.0 / n)
```

Python shell

```
| 1.0
| 1.25
| 1.328125
| 1.33331298828125
| 1.3333333330228925
| 1.3333333333333333
| Apx of 1.0 / 0.75 = 1.3333333333333333
| Python 1.0 / 0.75 = 1.3333333333333333
```

Operations

- None, bool
- basic operations
- strings
- += and friends

NoneType

- The type `None` has only one value: `None`
- Used when context requires a value, but none is really available
- **Example:** All functions must return a value. The function `print` has the *side-effect* of printing something to the standard output, but returns `None`
- **Example:** Initialize a variable with no value, e.g. list entries `mylist = [None, None, None]`

Python shell

```
> x = print(42)
| 42
> print(x)
| None
```

Type bool

- The type `bool` only has two values: `True` and `False`
- Logic truth tables:

<code>x or y</code>	True	False
True	True	True
False	True	False

<code>x and y</code>	True	False
True	True	False
False	False	False

<code>x</code>	<code>not x</code>
True	False
False	True

Scalar vs Non-scalar Types

- **Scalar types** (atomic/indivisible): `int`, `float`, `bool`, `None`
- **Non-scalar**: Examples `strings` and `lists`

```
"string"[3] = "i"  
[2, 5, 6, 7][2] = 6
```

Questions – What is $[7, 3, 5] [[1, 2, 3] [1]]$?

a) 1

b) 2

c) 3



d) 5

e) 7

f) Don't know

Operations on int and float

Result is float if and only if at least one argument is float, except `**` with negative exponent always gives a float

- `+`, `-`, `*` addition multiplication, e.g. $3.0 * 2 = 6.0$
- `**` and `pow(x, y)` power, e.g. $2 ** 3 = \text{pow}(2, 3) = 8$, $2 ** -2 = 0.25$
- `//` **integer division** = $\lfloor x / y \rfloor$
e.g. $15.0 // 4 = 3.0$. Note: $-8 // 3 = -3$
- `/` **division returns float**, $6 / 3 = 2.0$
- `abs(x)` absolute value
- `%` integer division remainder (modulo)
 $11 \% 3 = 2$
 $4.7 \% 0.6 = 0.500000000000000000003$

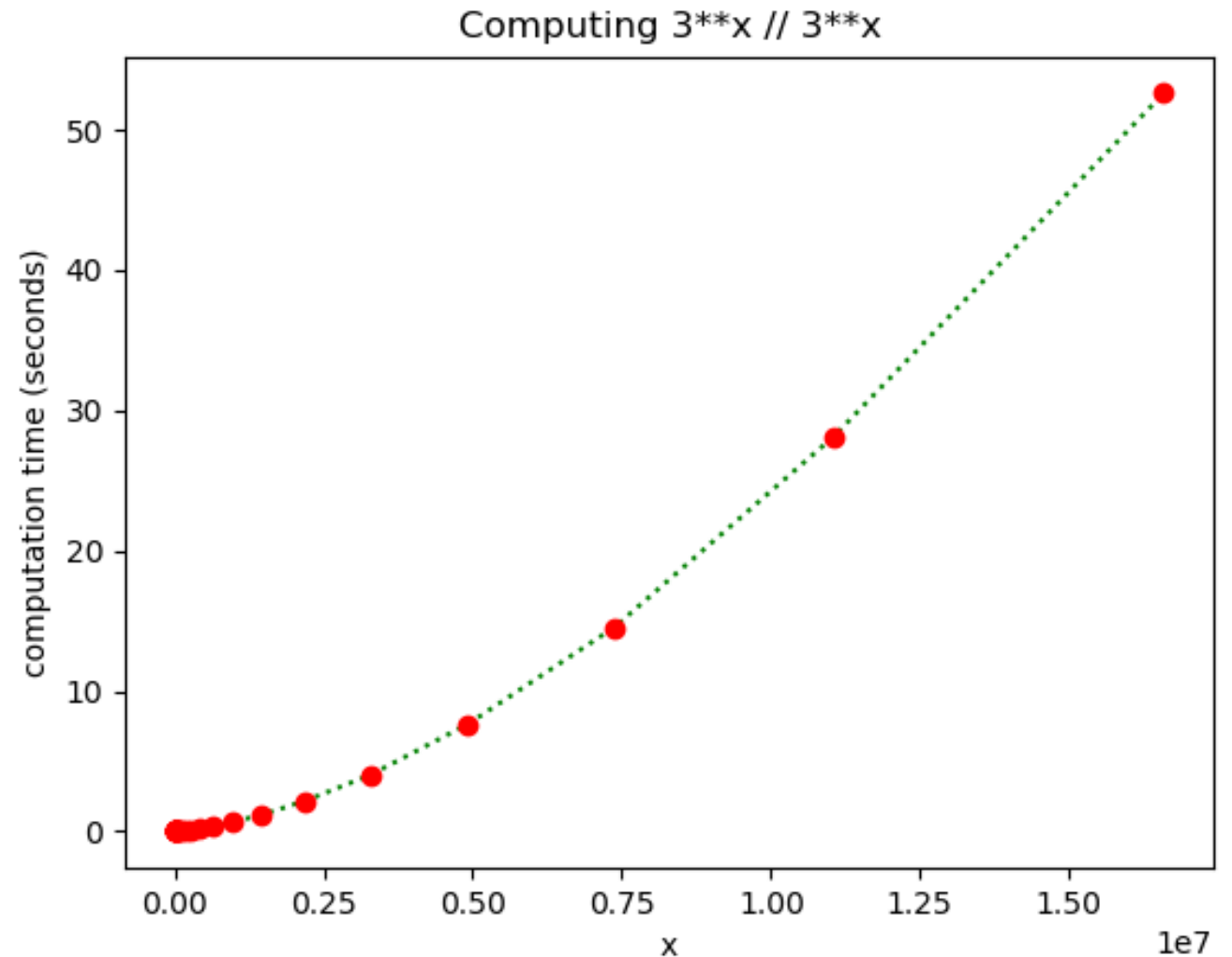
Python shell

```
> 0.4 // 0.1
| 4.0
> 0.4 / 0.1
| 4.0
> 0.3 // 0.1
| 2.0
> 0.3 / 0.1
| 2.9999999999999996
> 10**1000 / 2
| OverflowError: integer division
  result too large for a float
```



Running time for $3^{**}x // 3^{**}x$

Working with larger integers takes slightly more than linear time in the number of digits



integer-division-timing.py

```
from time import time
import matplotlib.pyplot as plt

bits, compute_time = [], []

for i in range(42):
    x = 3 ** i // 2 ** i
    start = time()
    result = 3 ** x // 3 ** x      # the computation we time
    end = time()
    t = end - start
    print('i =', i, 'x =', x, 'Result =', result, 'time(sec) =', t)
    bits.append(x)
    compute_time.append(t)

plt.title('Computing 3**x // 3**x')
plt.xlabel('x')
plt.ylabel('computation time (seconds)')
plt.plot(bits, compute_time, 'g:')
plt.plot(bits, compute_time, 'ro')
plt.show()
```

module math

Many standard mathematical functions are available in the Python module “`math`”, e.g.

`sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `log`(natural), `log10`, `exp`, `ceil`, `floor`, ...

- To use all the functions from the `math` module use `import math`
Functions are now available as e.g. `math.sqrt(10)` and `math.ceil(7.2)`
- To import selected functions you instead write `from math import sqrt, ceil`
- The library also contains some constants, e.g.
`math.pi` = 3.141592... and `math.e` = 2.718281...
- Note: `x ** 0.5` significantly faster than `sqrt(x)`



```
Python shell
> (0.1 + 0.2) * 10
| 3.0000000000000004
> math.ceil((0.1 + 0.2) * 10)
| 4
```


Python shell

```
> import math
> math.sqrt(8)
| 2.8284271247461903
> from math import pi, sqrt
> pi
| 3.141592653589793
> sqrt(5)
| 2.23606797749979
> from math import sqrt as kvadratrod
> kvadratrod(3)
| 1.7320508075688772

> import timeit
> timeit.timeit('1e10**0.5')
| 0.021124736888936863
> timeit.timeit('sqrt(1e10)', 'from math import sqrt')
| 0.1366314052865789
> timeit.timeit('math.sqrt(1e10)', 'import math')
| 0.1946660841634582
```

Rounding up integer fractions

- Python: $\lceil x/y \rceil = -(-x // y)$

-(-13/3)		
Python	Java	C
$-(-13//3)$ = 5	$-(-13/3)$ = 4 	$-(-13/3)$ = 4 



The intermediate result x/y in `math.ceil(x/y)`

is a float with limited precision

- Alternative computation:

$$\lceil x/y \rceil = (x + (y - 1)) // y$$

Python shell

```
> from math import ceil
> from timeit import timeit

> 13 / 3
| 4.333333333333333
> 13 // 3
| 4
> -13 // 3
| -5
> -(-13 // 3)
| 5
> ceil(13 / 3)
| 5

> -(-22222222222222222222223 // 2)
| 1111111111111111111112
> ceil(22222222222222222222223 / 2)
| 111111111111111111110656 
> timeit('ceil(13 / 3)', 'from math import ceil')
| 0.2774667127609973
> timeit('-(-13 // 3)') # negation trick is fast
| 0.05231945830200857
```

floats : Overflow, inf, -inf, nan

- There exists special float values
`inf`, `-inf`, `nan`
representing “+infinity”, “-infinity” and
“not a number”
- Can be created using e.g.
`float('inf')`
or imported from the `math` module
- Some overflow operations generate an
`OverflowError`, other return `inf`
and allow calculations to continue !
- Read the [IEEE 754 standard](#) if you want to
know more details...

Python shell

```
> 1e250 ** 2
| OverflowError:
|   (34, 'Result too large')
> 1e250 * 1e250
| inf
> -1e250 * 1e250
| -inf
> import math
> math.inf
| inf
> type(math.inf)
| <class 'float'>
> math.inf / math.inf
| nan
> type(math.nan)
| <class 'float'>
> math.nan == math.nan
| False
> float('inf') - float('inf')
| nan
```

Operations on bool

- The operations `and`, `or`, and `not` behave as expected when the arguments are `False/True`.
- The three operators also accept other types, where the following values are considered *false*:

`False, None, 0, 0.0, "", []`

(see The Python Standard Library > Built-in Types > [True Value Testing](#) for more *false* values)

- **Short-circuit evaluation**: The rightmost argument of `and` and `or` is only evaluated if the result cannot be determined from the leftmost argument alone. The result is either the leftmost or rightmost argument (see truth tables), i.e. the result is not necessarily `False/True`.

`True or 7/0` is completely valid since `7/0` will never be evaluated
(which otherwise would throw a `ZeroDivisionError` exception)

x	x or y
<i>false</i>	y
otherwise	x

x	x and y
<i>false</i>	x
otherwise	y

x	not x
<i>false</i>	True
otherwise	False

Questions – What is "abc" and 42 ?

a) False

b) True

c) "abc"



d) 42

e) TypeError

f) Don't know

Comparison operators (e.g. int, float, str)

`==` test if two objects are equal, returns bool
not to be confused with the assignment operator (`=`)

`!=` not equal

`>`

`>=`

`<`

`<=`

Python shell

```
> 3 == 7
| False
> 3 == 3.0
| True
> "-1" != -1
| True
> "abc" == "ab" + "c"
| True
> 2 <= 5
| True
> -5 > 5
| False
> 1 == 1.0
| True
> 1 == 1.000000000000000001
| True
> 1 == 1.000000000000000001
| False
```



Chained comparisons

- A recurring condition is often

$$x < y \text{ and } y < z$$

- If y is a more complex expression, we would like to avoid computing y twice, i.e. we often would write

```
tmp = complex expression
x < tmp and tmp < z
```

- In Python this can be written as a **chained comparisons** (which is shorthand for the above)

$$x < y < z$$

- Note: Chained comparisons do not exist in C, C++, Java, ...

Questions – What is $1 < 0 < 6/0$?

- a) True
-  b) False
- c) 0
- d) 1
- e) 6
- f) ZeroDivisionError
- g) Don't know

Binary numbers and operations

- Binary number = integer written in base 2: $101010_2 = 42_{10} = 2^1 + 2^3 + 2^5$
- Python constant prefix 0b: `0b101010` → 42
- `bin(x)` converts integer to string: `bin(49)` → "0b110001"
- `int(x, 2)` converts binary string value to integer: `int("0b110001", 2)` → 49
- Bitwise operations
 - | Bitwise OR
 - & Bitwise AND
 - ~ Bitwise NOT (~ x equals to -x - 1)
 - ^ Bitwise XOR
- Example: `bin(0b1010 | 0b1100)` → "0b1110"
- Hexadecimal = base 16, Python prefix 0x: `0x30` → 48, `0xA0` → 160, `0xFF` → 255
- << and >> integer bit shifting left and right, e.g. `12 >> 2` → 3, and `1 << 4` → 16

Operations on strings

- `len(str)` returns length of `str`
- `str[index]` returns `index+1`'th symbol in `str`
- `str1 + str2` returns concatenation of two strings
- `int * str` concatenates `str` with itself `int` times
- Formatting: % operator or .format() function
old Python 2 way since Python 3.0
or formatted string literals (f-strings) with prefix
since Python 3.6
letter `f` and Python expressions in `{ }`
(see pyformat.info for an introduction)

From “[What’s New In Python 3.0](#)”, 2009: A new system for built-in string formatting operations replaces the % string formatting operator. (However, *the % operator* is still supported; it *will be deprecated in Python 3.1* and removed from the language at some later time.) Read [PEP 3101](#) for the full scoop.

Python shell

```
> len("abcde")
| 5
> "abcde"[2]
| 'c'
> x, y = 2, 3
> "x = %s, y = %s" % (x, y)
| 'x = 2, y = 3'
> "x = {}, y = {}".format(x, y)
| 'x = 2, y = 3'
> "x = {1}, y = {0}".format(y, x)
| 'x = 2, y = 3'
> f'x + y = {x + y}'
| 'x + y = 5'
> f'{x + y = }' # >= Python 3.8
| 'x + y = 5'
> f'{x} / {y} = {x / y:.3}'
| '2 / 3 = 0.667'
> "abc" + "def"
| 'abcdef'
> 3 * "x--"
| 'x--x--x--'
> 0 * "abc"
| ''
```

% formatting (inherited from C's `sprint()` function) was supposed to be on the way out - but is still going strong in Python 3.13

Python does not automatic convert to string



- A `str` can not be concatenated with an `int`
- The `int` must first be converted to a `str` using `str(int value)`

works in Java
and Javascript

Python shell

```
> 'abc' + 42
| TypeError: can only concatenate str (not "int") to str
> 'abc' + str(42)
| 'abc42'
```

- In some languages, like Java and Javascript, the conversion to string happens automatically (just a language design choice)

... more string functions

- `str[-i]` returns the symbol *i* positions from the right, the rightmost `str[-1]`
- `str[from:to]` substring starting at index *from* and ending at index *to*-1
- `str[from:-to]` substring starting at *from* and last at index `len(str) - to - 1`
- `str[from:to:step]` only take every *step*'th symbol in `str[from:to]`
 - *from* or/and *to* can be omitted and defaults to the beginning/end of string
- `chr(x)` returns a string of length 1 containing the *x*'th Unicode character
- `ord(str)` for a string of length 1, returns the Unicode number of the symbol
- `str.lower()` returns string in lower case
- `str.split()` split string into list of words, e.g.
`"we love python".split() = ['we', 'love', 'python']`

Python shell

```
> s = 'this is a string'
> s.capitalize()
| 'This is a string'
> s.title()
| 'This Is A String'
> s.upper()
| 'THIS IS A STRING'
> s.title().swapcase()
| 'tHIS iS a sTRING'
> s.removeprefix('this is ')
| 'a string'
> s.removesuffix(' string')
| 'this is a'
> s.replace('is', 'IS')
| 'thIS IS a string'
> s.ljust(30)
| 'this is a string          '
> s.center(30)
| '      this is a string      '
> s.rjust(30, '.')
| '.....this is a string'
> s.count('i')
| 3
> '12345'.zfill(8)
| '00012345'
```

Questions – What is `s[2:42:3]`?

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
s = 'abwwdexy__lwtopavghevt_xypxxyattx_hxwoadnxxx'

- a) 'wwdexy____lwttopavghevt_xypxxyattx_hxwoadn'
-  b) 'we_love_python'
- c) 'we_love_java'
- d) Don't know

Strings are immutable

- Strings are non-scalar, i.e. for `s = "abcdef"`, `s[3]` will return `"d"`
- Strings are **immutable** and cannot be changed once created.
I.e. the following natural update **is not possible** (but is e.g. allowed in C)

`s[3] = "x"`

- To replace the `"d"` with `"x"` in `s`, instead create the new string

`s = s[:3] + "x" + s[4:]`

Operators

Precedence rules & Associativity

Example: * has higher precedence than +

$$2 + 3 * 4 \equiv 2 + (3 * 4) \rightarrow 14 \quad \text{and} \quad (2 + 3) * 4 \rightarrow 20$$

All operators in same group are evaluated left-to-right

$$2 + 3 - 4 - 5 \equiv ((2 + 3) - 4) - 5 \rightarrow -4$$

except for **, that is evaluated right-to-left

$$2 ** 2 ** 3 \equiv 2 ** (2 ** 3) \rightarrow 256$$

Rule: Use **parenthesis** whenever in doubt of precedence!

Precedence (low to high)		
or		
and		
not x		
in	not in	
is	is not	
==	<	<=
!=	>	>=
^		
&		
<<		>>
+		-
*		@
/	//	%
+x	-x	~x
**		

Long expressions

- Long expressions can be broken over several lines by putting parenthesis around it
- The PEP8 guidelines recommend to limit **all** lines to a maximum of 79 characters

```
Python shell
> (1
    + 2 +
    3)
| 6
```

+= and friends

- Recurring statement is

`x = x + value`

- In Python (and many other languages) this can be written as

`x += value`

- This also applies to other operators like

`+=    -=    *=    /=    //=    **=`
`|=    &=    ^=    <<=    >>=`

Python shell

```
> x = 5
> x *= 3
> x
| 15
> a = 'abc'
> a *= 3
> a
| 'abcabcabc'
```

`:=` assignment expressions (the “Walrus Operator”)

New in
Python 3.8

- Syntax

`name := expression`

- Evaluates to the value of `expression`, with the side effect of assigning result to `name`
- Useful for naming intermediate results/repeating subexpressions for later reuse
- See [PEP 572](#) for further details and restrictions of usage

Python shell

```
> (x := 2 * 3) + 2 * x
| 18
> print(1 + (x := 2 * 3), 2 + x)
| 7 8
> x := 7
| SyntaxError
> (x := 7) # valid, but not recommended
> while line := input():
|     print(line.upper())
> abc
| ABC
```

- In some languages, e.g. Java, C and C++, “=” also plays the role of “:=”, implying “if (x=y)” and “if (x==y)” mean quite different things (common typo)



Lists

- List syntax
- List operations
- `copy.deepcopy`
- `range`
- `while-else`
- `for`
- `for-break-continue-else`



`L[-index] == L[len(L)-index]`
Negative indexing from the right is Python specific, and is e.g. not supported in Java, C, C++

List operations

- List syntax `[value0, value1, ..., valuek-1]`
- List indexing `L[index]`, `L[-index]` ←
- ★ ■ List **slices** `L[from:to]`, `L[from:to:step]` or `L[slice(from, to, step)]`
- ★ ■ Creating a copy of a list `L[:]` or `L.copy()`
- ★ ■ List concatenation (creates new list) `X + Y`
- ★ ■ List repetition (repeated concatenation with itself) `42 * L`
- Length of list `len(L)`
- Check if element is in list `e in L` (returns True or False)
- Check if element is not in list `e not in L` (same as `not e in L`)
- Index of first occurrence of element in list `L.index(e)`
- Number of occurrences of element in list `L.count(e)`
- `sum(L)` `min(L)` `max(L)`
- ★ = creates new list

sum (...)



Python shell

```
> 1 - 1/3 - 1 + 1/3 # mathematically should be zero
| 5.551115123125783e-17 # but floats are imprecise
> L = [1, -1/3, -1, 1/3]
> L
| [1, -0.3333333333333333, -1, 0.3333333333333333] # mix of int and float
> sum(L)
| 5.551115123125783e-17
> sum([1.0, -1/3, -1.0, 1/3]) # all floats
| 5.551115123125783e-17 # Python 3.11
> sum([1.0, -1/3, -1.0, 1/3])
| 0.0 # Python 3.12 uses "Neumaier summation" to improve accuracy for floats
> sum([1, -1/3, -1, 1/3])
| 5.551115123125783e-17 # Python 3.12 loses accuracy when mixing int and float
> import math
> math.fsum([1, -1/3, -1, 1/3]) # math.fsum more accurate float sums
| 0.0
```



List modifiers (lists are mutable)

- Extend list with elements (X is modified) `X.extend(Y)`
- Append an element to a list (L is modified) `L.append(42)`
- Replace sublist by another list (length can differ) `X[i:j] = Y`
- Delete elements from list `del L[i:j:k]`
- Remove & return element at position `L.pop(i)`
- Remove first occurrence of element `L.remove(e)`
- Reverse list `L.reverse()`
- `L *= 42`
- `L.insert(i, x)` same as `L[i:i] = [x]`

Python shell

```
> x = [1, 2, 3, 4, 5]
> x[2:4] = [10, 11, 12]
> x
| [1, 2, 10, 11, 12, 5]
> x = [1, 2, 11, 5, 8]
> x[1:4:2] = ['a', 'b']
| [1, 'a', 11, 'b', 8]
```

Questions – What is `x` ?

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
x[2:8:3] = ['a', 'b']
```

- a) `[1, 2, 'a', 'b', 5, 6, 7, 8, 9, 10]`
- b) `[1, 'a', 3, 4, 5, 6, 7, 'b', 9, 10]`
- c) `[1, 2, 3, 4, 5, 6, 7, 'a', 'b']`
-  d) `[1, 2, 'a', 4, 5, 'b', 7, 8, 9, 10]`
- e) `ValueError`
- f) Don't know

Questions – What is y ?

```
y = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
y = y[3:15:3][1:4:2]
```

- a) [3, 6, 9, 12, 15]
-  b) [7, 13]
- c) [1, 9]
- d) [4, 7, 10, 13, 2, 4]
- e) TypeError
- f) Don't know

Nested lists (multi-dimensional lists)

- Lists can contain lists as elements, that can contain lists as elements, that ...
- Can e.g. be used to store multi-dimensional data (list lengths can be non-uniform)

 **Note:** For dealing with matrices the NumPy module is a better choice

multidimensional-lists.py

```
list1d = [1, 3, 5, 2]
list2d = [[1, 2, 3, 4],
          [5, 6, 7, 9],
          [0, 8, 2, 3]]
list3d = [[[5,6], [4,2], [1,7], [2,4]],
          [[1,2], [6,3], [2,5], [7,5]],
          [[3,8], [1,5], [4,3], [2,4]]]

print(list1d[2])
print(list2d[1][2])
print(list3d[2][0][1])
```

Python shell

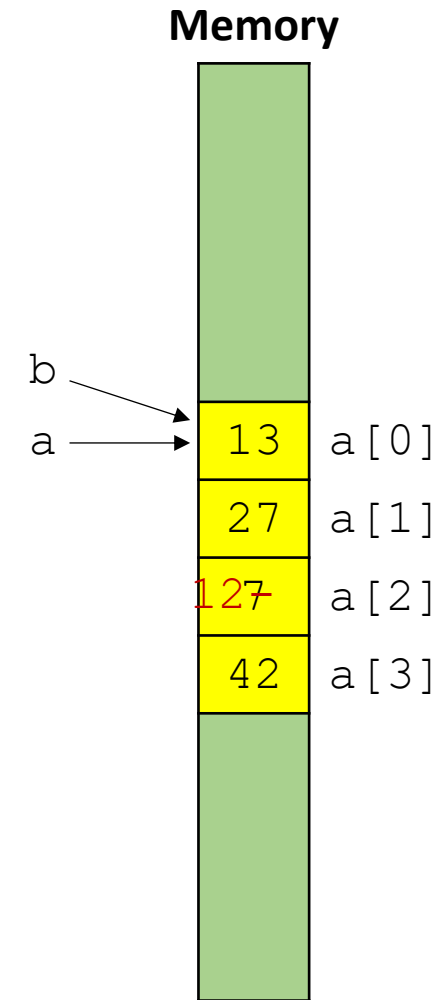
```
| 5
| 7
| 8
```

aliasing

```
a = [13, 27, 7, 42]
```

```
b = a
```

```
a[2] = 12
```

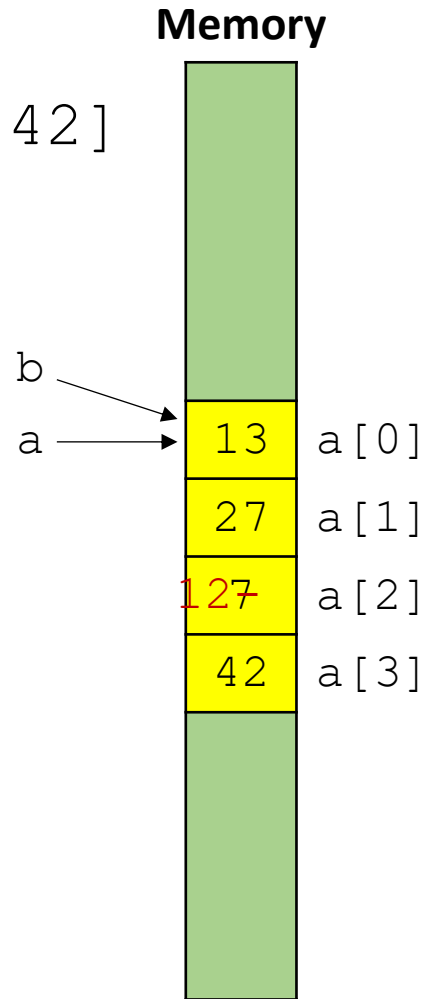


$y = x$ VS $y = x[:]$

`a = [13, 27, 7, 42]`

`b = a`

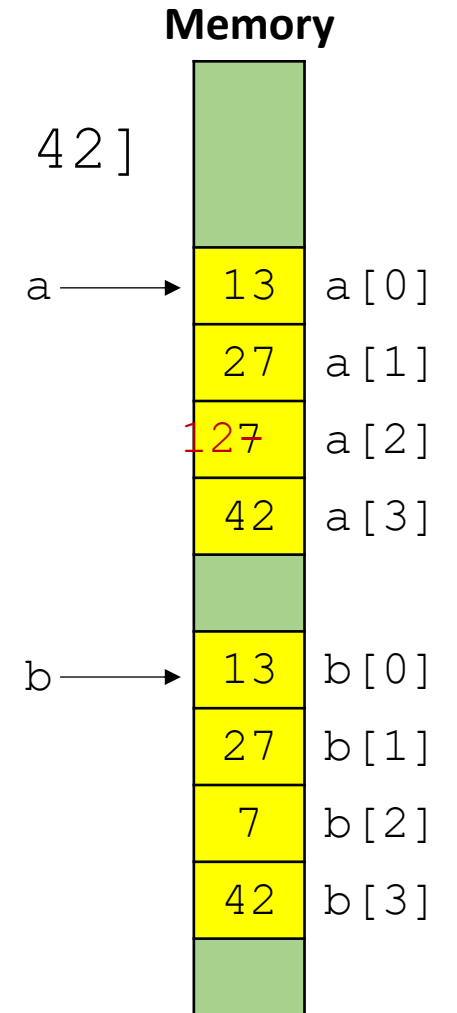
`a[2] = 12`



`a = [13, 27, 7, 42]`

`b = a[:]`

`a[2] = 12`



× [:] vs nested structures

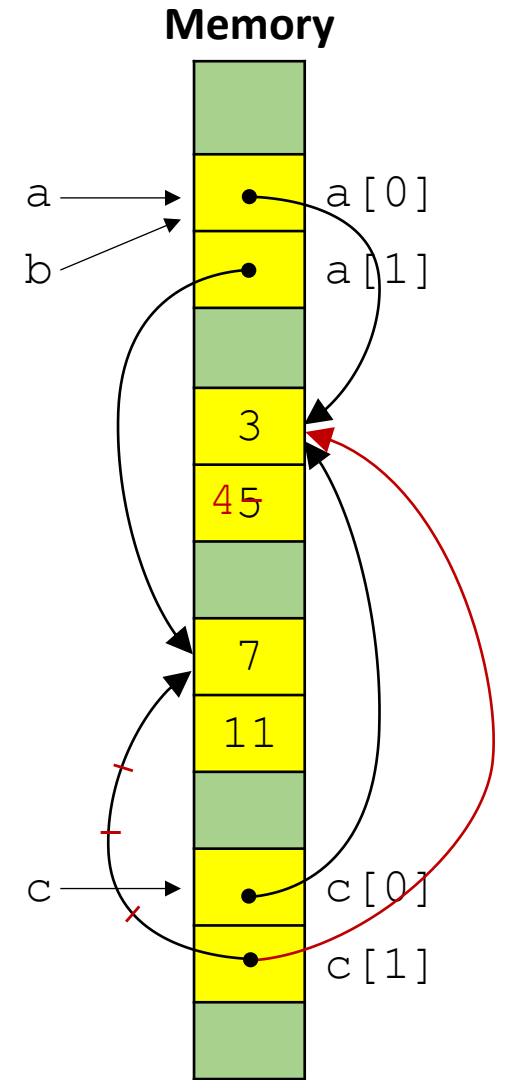
```
a = [[3, 5], [7, 11]]
```

```
b = a
```

```
c = a[:]
```

```
a[0][1] = 4
```

```
c[1] = b[0]
```



Question – what is c ?

- a) `[[3, 5], [7, 11]]`
- b) `[[3, 5], [3, 5]]`
- c) `[[3, 4], [3, 5]]`
- 😊 d) `[[3, 4], [3, 4]]`
- e) Don't know

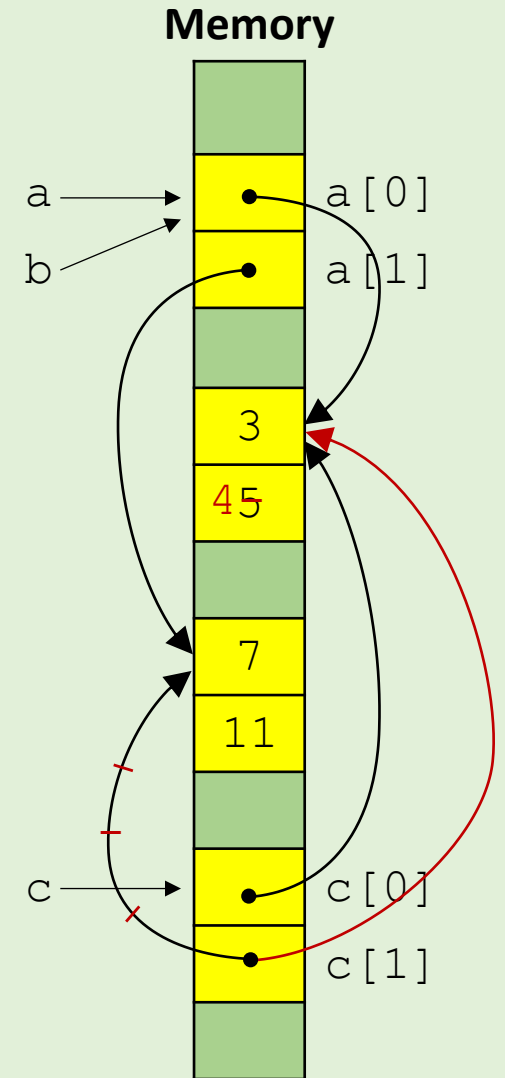
```
a = [[3, 5], [7, 11]]
```

```
b = a
```

```
c = a[:]
```

```
a[0][1] = 4
```

```
c[1] = b[0]
```

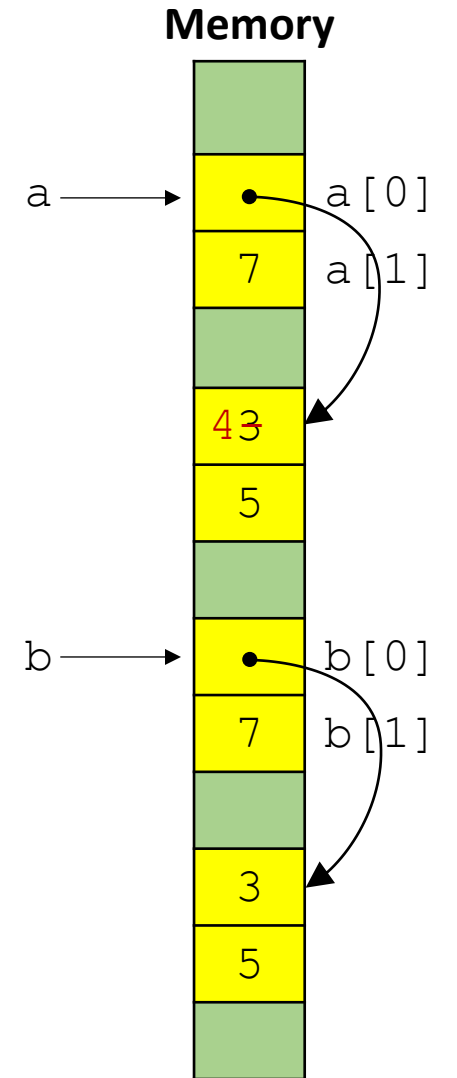


copy.deepcopy

- To make a copy of all parts of a composite value use the function **deepcopy** from module copy

Python shell

```
> from copy import deepcopy
> a = [[3, 5], 7]
> b = deepcopy(a)
> a[0][0] = 4
> a
| [[4, 5], 7]
> b
| [[3, 5], 7]
```



Initializing a 2-dimensional list



Python shell

```
> x = [1] * 3
> x
| [1, 1, 1]
> y = [[1] * 3] * 4
> y
| [[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]
> y[0][0] = 0
> y
| [[0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1]]
```

Python shell

```
> y = []
> for _ in range(4): y.append([1] * 3)
> y[0][0] = 0
> y
| [[0, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]
```


range(from, to, step)



In Python 2, `range` generates the explicit list, i.e. always use memory proportional to the length; `xrange` in Python 2 corresponds to `range` in Python 3; Python 3 is more memory friendly

- `range(from, to, else)` represents the **sequence** of numbers starting with *from*, with increments of *step*, and smaller/greater than *to* if *step* positive/negative

```
range(5)           : 0, 1, 2, 3, 4   (default from = 0, step = 1)
range(3, 8)        : 3, 4, 5, 6, 7   (default step = 1)
range(-2, 7, 3)     : -2, 1, 4       (from and to can be any integer)
range(2, -5, -2)    : 2, 0, -2, -4   (decreasing sequence if step negative)
```

- Ranges are immutable, can be indexed like a list, sliced, and compared (i.e. generate the same numbers)
- `list(range(...))` generates the explicit list of numbers

Python shell

```
> range(1, 10000000, 3)[2]
| 7
> range(1, 10000000, 3)[100:120:4]
| range(301, 361, 12)
> range(1, 10000000, 3)[100:120:4][2:3]
| range(325, 337, 12)
> list(range(5, 14, 3))
| [5, 8, 11]
```

Question – What is `range(3, 20, 4)[2:4][1]` ?

a) 3

b) 7

c) 11

 d) 15

e) 19

f) Don't know

for - loop

- For every element in a **sequence** execute a block of code:

```
for var in sequence:  
    block
```

- Sequences can e.g. be lists, strings, ranges
- `break` and `continue` can be used like in a while-loop to break out of the for-loop or continue with the next element in the sequence

Python shell

```
> for x in [1, 'abc', [2, 3], 5.0]:  
>     print(x)  
| 1  
| abc  
| [2, 3]  
| 5.0  
> for x in 'abc':  
>     print(x)  
| a  
| b  
| c  
> for x in range(5, 15, 3):  
>     print(x)  
| 5  
| 8  
| 11  
| 14
```

Question – What is printed ?

Python shell

```
> for i in range(1, 4):  
>     for j in range(i, 4):  
>         print(i, j, sep=':', end=' ')
```

a) 1:1 1:2 1:3 2:1 2:2 2:3 3:1 3:2 3:3

 b) 1:1 1:2 1:3 2:2 2:3 3:3

c) 1:1 2:1 3:1 1:2 2:2 3:2 1:3 2:3 3:3

d) 1:1 2:1 3:1 2:2 3:2 3:3

e) Don't know

Question – **break**, what is printed ?

Python shell

```
> for i in range(1, 4):  
>     for j in range(1, 4):  
>         print(i, j, sep=':', end=' ')  
>         if j >= i:  
>             break
```

- a) *nothing*
- b) 1:1
-  c) 1:1 2:1 2:2 3:1 3:2 3:3
- d) 1:1 2:2 3:3
- e) Don't know



In nested for- and while-loops,
break only breaks the innermost loop

***** COMMODORE 64 BASIC V2 *****

64K RAM SYSTEM 38911 BASIC BYTES FREE

READY.

10 FOR I=1 TO 10

20 PRINT I

30 NEXT I

RUN

1

2

3

4

5

6

7

8

9

10

READY.

█

***** COMMODORE 64 BASIC V2 *****

64K RAM SYSTEM 38911 BASIC BYTES FREE

```
READY.  
10 FOR I=1 TO 3  
20   FOR J=1 TO 3  
30     PRINT I,J  
40   NEXT J  
50 NEXT I  
RUN
```

1
1
1
2
3

1
2
3
1
2
3

READY.

Palindromic substrings

- Find all **palindromic** substrings of length ≥ 2 , i.e. substrings spelled identically forward and backwards:

abracadbratrallalla

i j i j

- Algorithm:** Test all possible substrings (brute force/exhaustive search)
- Note:** the slice `t[::-1]` is `t` reversed

palindrom.py

```
s = 'abracadbratrallalla'

for i in range(len(s)):
    for j in range(i + 2, len(s) + 1):
        t = s[i:j]
        if t == t[::-1]:
            print(t)
```

Python shell

```
| aca
| alla
| allalla
| ll
| lla11
| la1
| alla
| ll
```


Sieve of Eratosthenes

- Find all prime numbers $\leq n$
- Algorithm:

② 3 4 5 6 7 8 9 10 11 12 13 14 ...
2 ③ 4 5 6 7 8 9 10 11 12 13 14 ...
2 3 4 ⑤ 6 7 8 9 10 11 12 13 14 ...
2 3 4 5 6 ⑦ 8 9 10 11 12 13 14 ...
2 3 4 5 6 7 8 9 10 ⑪ 12 13 14 ...
2 3 4 5 6 7 8 9 10 11 12 ⑬ 14 ...

eratosthenes.py

```
n = 100
prime = [True] * (n + 1)

for i in range(2, n):
    for j in range(2 * i, n + 1, i):
        prime[j] = False

for i in range(2, n + 1):
    if prime[i]:
        print(i, end=' ')
```

Python shell

```
| 2 3 5 7 11 13 17 19 23 29 31 37 41
  43 47 53 59 61 67 71 73 79 83 89
  97
```

while-else and for-else loops

- Both for- and while-loops can have an optional “else”:

```
for var in sequence:  
    block  
else:  
    block
```

```
while condition:  
    block  
else:  
    block
```

- The “else” block is only executed if no `break` is performed in the loop
- The “else” construction for loops is specific to Python, and does not exist in e.g. C, C++ and Java



Linear search

linear-search-while.py

```
L = [7, 3, 6, 4, 12, 'a', 8, 13]
x = 4

i = 0
while i < len(L):
    if L[i] == x:
        print(x, 'at position', i, 'in', L)
        break
    i = i + 1

if i >= len(L):
    print(x, 'not in', L)
```

linear-search-while-else.py

```
i = 0
while i < len(L):
    if L[i] == x:
        print(x, 'at position', i, 'in', L)
        break
    i = i + 1
else:
    print(x, 'not in', L)
```

linear-search-for.py

```
found = False
for i in range(len(L)):
    if L[i] == x:
        print(x, 'at position', i, 'in', L)
        found = True
        break

if not found:
    print(x, 'not in', L)
```

linear-search-for-else.py

```
for i in range(len(L)):
    if L[i] == x:
        print(x, 'at position', i, 'in', L)
        break
else:
    print(x, 'not in', L)
```

linear-search-builtin.py

```
if x in L:
    print(x, 'at position', L.index(x), 'in', L)
else:
    print(x, 'not in', L)
```

Some performance considerations

String concatenation

- To concatenate two (or few) strings use

$str_1 + str_2$
 $var += str$

- To concatenate several/many strings use

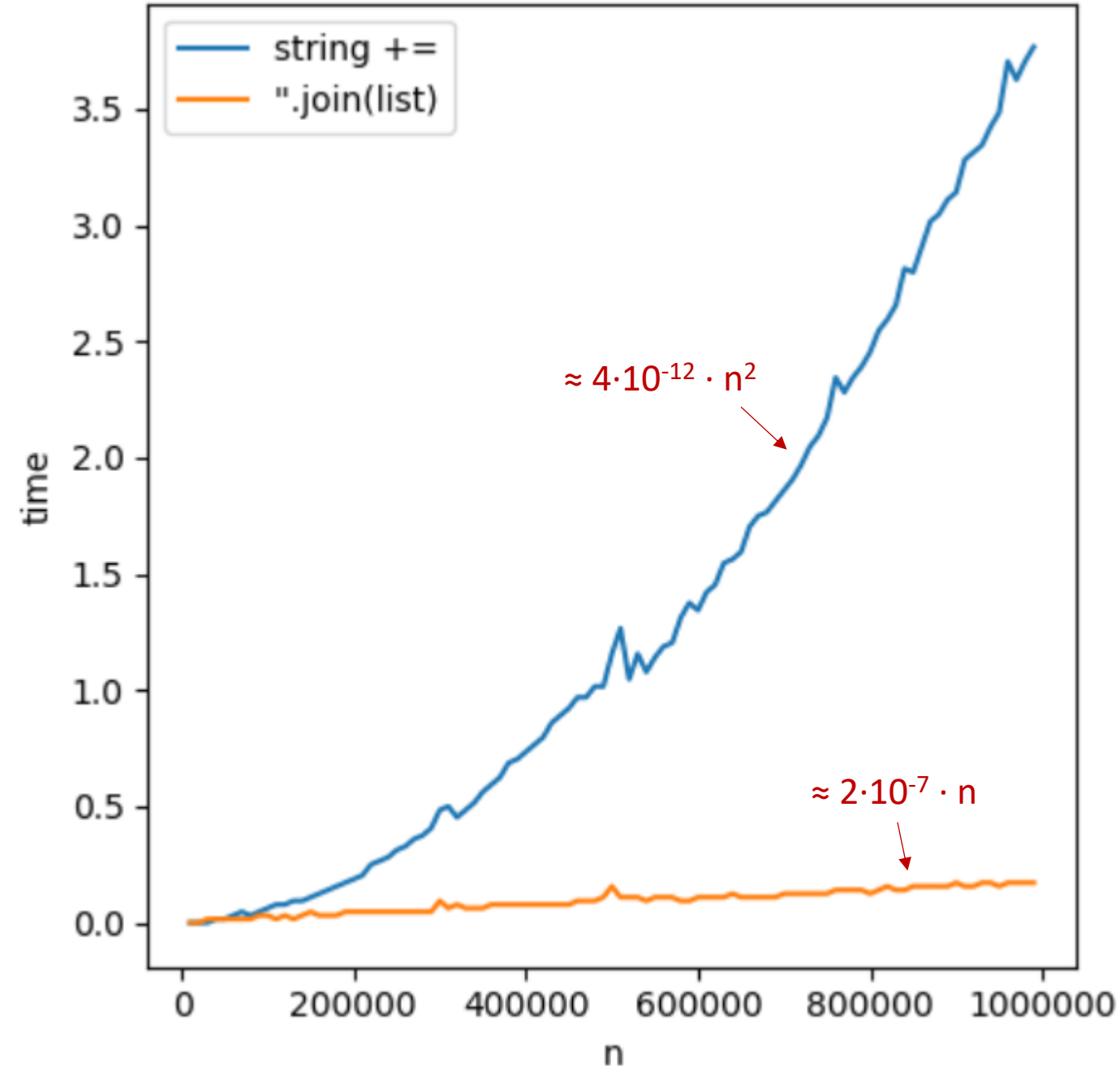
`' '.join([str1, str2, str3, ... , strn])`

- Concatenating several strings by repeated use of `+` generates explicitly the longer-and-longer intermediate results; using `join` avoids this slowdown



Python shell

```
> s = 'A' + 'B' + 'C'
> s
| 'ABC'
> 'x'.join(['A', 'B', 'C'])
| 'AxBxC'
> s = ''
> s += 'A'
> s += 'B'
> s += 'C'
> s
| 'ABC'
> L = []
> L.append('A')
> L.append('B')
> L.append('C')
> L
| ['A', 'B', 'C']
> s = ''.join(L)
> s
| 'ABC'
```



string-concatenation.py

```
from time import time
from matplotlib import pyplot as plt

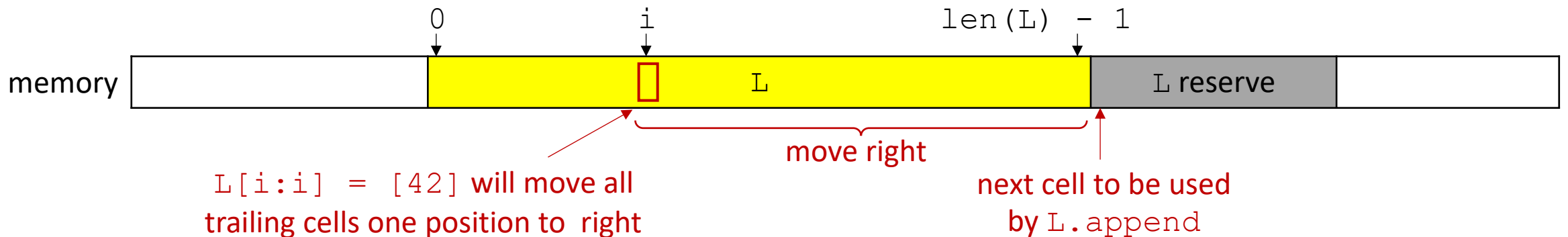
ns = range(10_000, 1_000_000, 10_000)
time_string = []
time_list = []
for n in ns:
    start = time()
    [
        s = ''
        for _ in range(n):
            s += 'abcdefgh' # slow ⚠
    ]
    end = time()
    time_string.append(end - start)

    start = time()
    [
        substrings = []
        for _ in range(n):
            substrings.append('abcdefgh')
    ]
    s = ''.join(substrings)
    end = time()
    time_list.append(end - start)

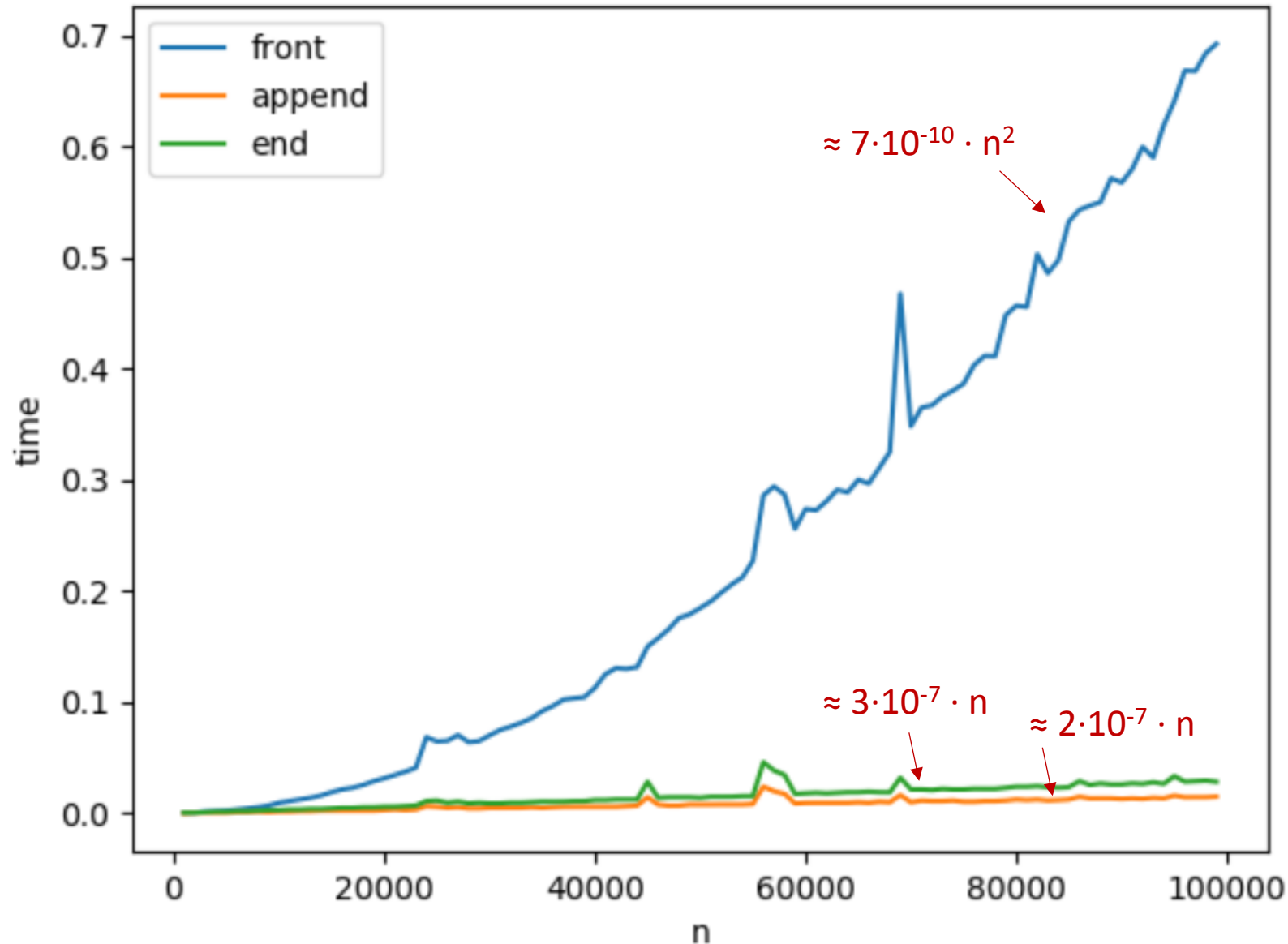
plt.plot(ns, time_string, label='string +=')
plt.plot(ns, time_list, label=" ''.join(list) ")
plt.xlabel('n')
plt.ylabel('time')
plt.legend()
plt.show()
```

The internal implementation of Python lists

- Accessing and updating list positions take the same time independently of position
- Creating new / deleting entries in a list depends on position, Python optimizes towards updates at the end
- Try to organize your usage of lists to insert / delete elements at the end
`L.append(element)` and `L.pop()`
- Python lists internally have space for adding $\approx 12.5\%$ additional entries at the end; when the reserved extra space is exhausted the list is moved to a new chunk of memory with $\approx 12.5\%$ extra space



List insertions at front vs end



list-insertions.py

```
from time import time
from matplotlib import pyplot as plt

ns = range(1000, 100_000, 1000)
time_end = []
time_append = []
time_front = []
for n in ns:
    start = time()
    L = []
    for i in range(n):
        L[i:i] = [i] # insert after list
    end = time()
    time_end.append(end - start)

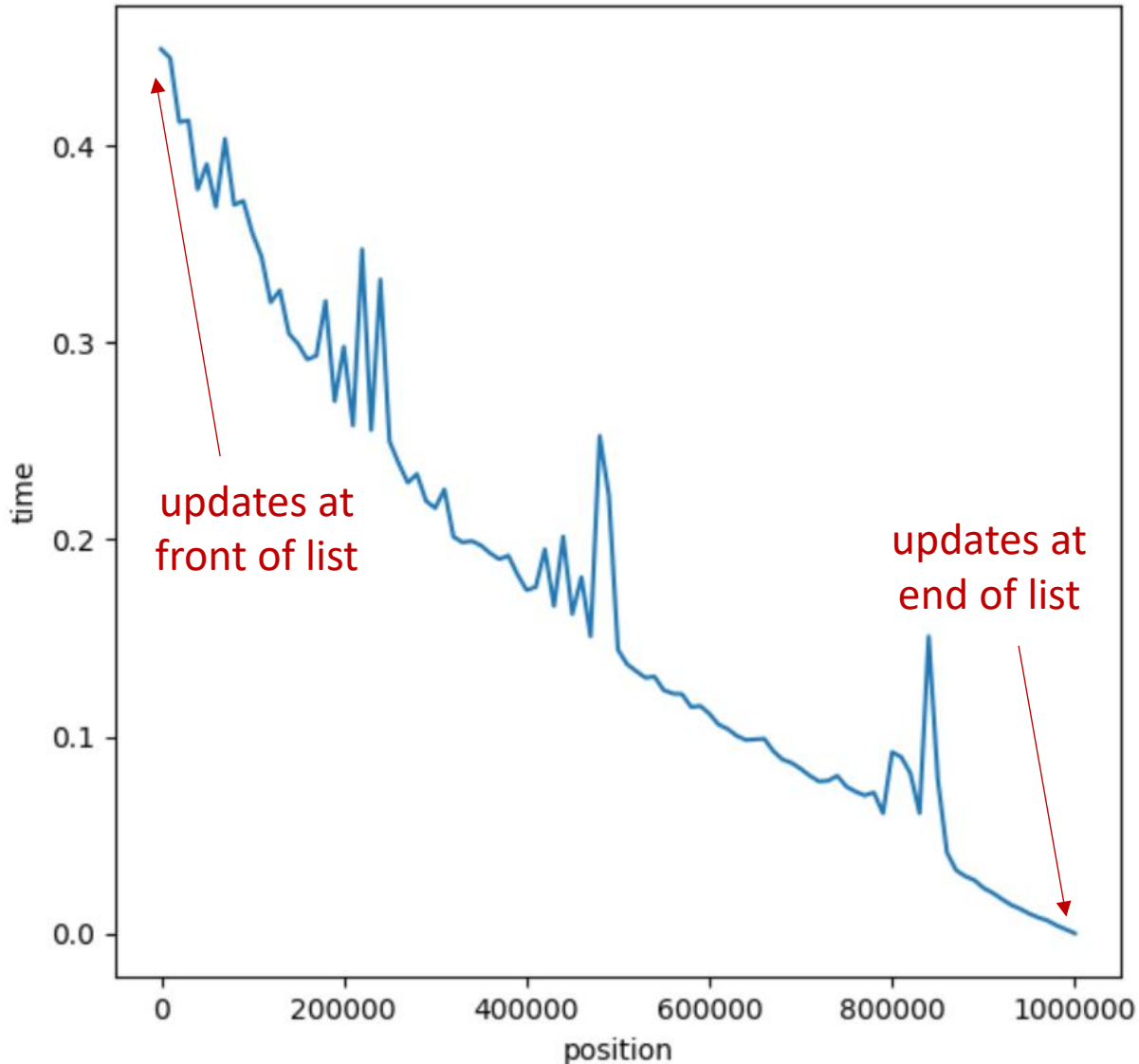
    start = time()
    L = []
    for i in range(n):
        L.append(i) # append to list
    end = time()
    time_append.append(end - start)

    start = time()
    L = []
    for i in range(n):
        L[0:0] = [i] # insert at front
    end = time()
    time_front.append(end - start)

plt.plot(ns, time_front, label='front')
plt.plot(ns, time_append, label='append')
plt.plot(ns, time_end, label='end')
plt.xlabel('n')
plt.ylabel('time')
plt.legend()
plt.show()
```



Updates (insertions + deletions) in the middle of a list



list-updates.py

```
from time import time
from matplotlib import pyplot as plt

ns = range(0, 1_000_001, 10_000)
time_pos = []
L = list(range(1_000_000)) # L = [0, ..., 999_999]
for i in ns:
    start = time()
    for _ in range(1000):
        L[i:i] = [42] # insert element before L[i]
        del L[i]      # remove L[i] from L
    end = time()
    time_pos.append(end - start)

plt.plot(ns, time_pos)
plt.xlabel('position')
plt.ylabel('time')
plt.show()
```

Tuples and lists

- tuples
- lists
- mutability
- list comprehension
- for-if, for-for
- list()
- any(), all()
- enumerate(), zip()

Tuples

$(\text{value}_0, \text{value}_1, \dots, \text{value}_{k-1})$

- Tuples can contain a sequence of zero or more elements, enclosed by " (" and ") "
- Tuples are **immutable**
- Tuple of length 0: `()`
- Tuple of length 1: `(value, )`
Note the **comma** to make a tuple of length one distinctive from an expression in parenthesis
- In many contexts a tuple with ≥ 1 elements can be written without parenthesis
- Accessors to lists also apply to tuples, slices, ...

Python shell

```
> (1, 2, 3)
| (1, 2, 3)
> ()
| ()
> (42)
| 42
> (42,)
| (42,)
> 1, 2
| (1, 2)
> 42,
| (42,)
> x = (3, 7)
> x
| (3, 7)
> x = 4, 6
> x
| (4, 6)
> x[1] = 42
| TypeError: 'tuple' object does not support item assignment
```

Question – What value is $((42,))$?

a) 42

b) (42)



c) (42,)

d) ((42,),)

e) Don't know

Warning: Unintended tuples



Python shell

```
> x = 7,      # x = (7, )  
> y = 5,      # y = (5, )  
> x + y  
| (7, 5)
```

Question – What is `x` ?

```
x = [1, [2, 3], (4, 5)]  
x[2][0] = 42
```

a) `[1, [42, 3], (4, 5)]`

b) `[1, [2, 3], (42, 5)]`

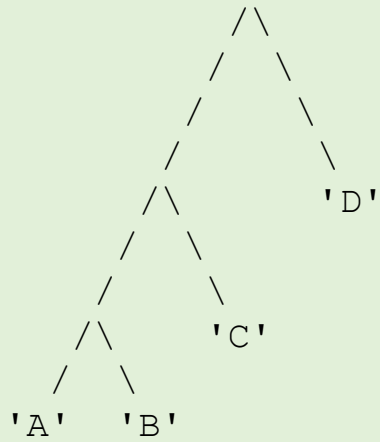
c) `[1, [2, 3], 42]`



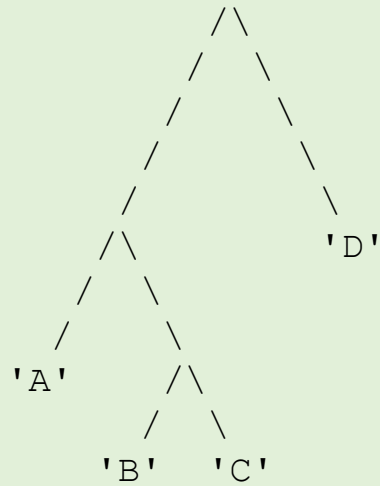
d) `TypeError`

e) Don't know

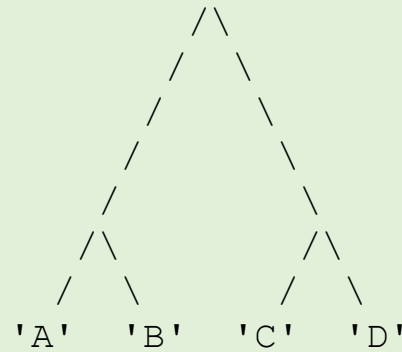
Question – What tree is $('A', (('B', 'C'), 'D'))$?



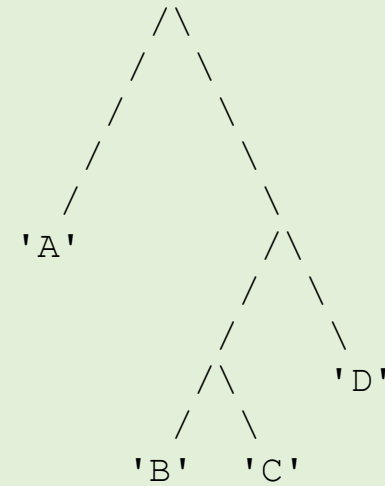
a)



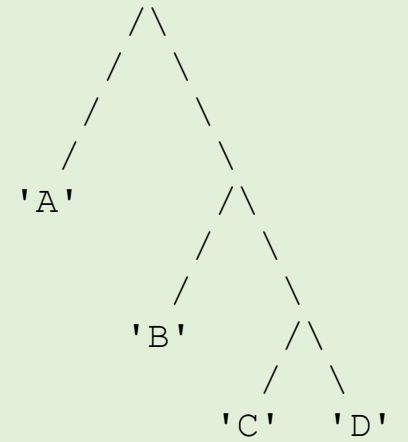
b)



c)



d)



e)

f) Don't know

Tuple assignment

Unpacking

Python shell

```
> point = (10, 25)
> x, y = point
> x
| 10
> y
| 25
```

- Parallel assignments

$$x, y, z = a, b, c$$

is a shorthand for a tuple assignment (right side is a single tuple)

$$(x, y, z) = (a, b, c)$$

- First the right-hand side is evaluated completely, and then the individual values of the tuple are assigned to x, y, z left-to-right (length must be equal on both sides)

Nested tuple/lists assignments

- Let hand side can be nested
(great for unpacking data)

```
(x, (y, (a[0], w)), a[1])  
= 1, (2, (3, 4)), 5
```

- [...] and (...) on left side matches both
lists and tuples of equal length
(but likely you would like to be
consistent with type of parenthesis)

Python shell

```
> two_points = [(10, 25), (30, 40)]  
> (x1, y1, x2, y2) = two_points  
| ValueError: not enough values to  
| unpack (expected 4, got 2)  
> ((x1, y1), (x2, y2)) = two_points  
> a = [None, None]  
> v = ((2, (3, 4)), 5)  
> ((y, (a[0], w)), a[1]) = v  
> a  
| [3, 5]  
> [x, y, z] = (3, 5, 7)  
> (x, y, z) = [3, 5, 7]  
> [x, (y, z), w] = (1, [2, 3], 4)  
> [x, (y, z), w] = (1, [2, (5, 6)], 4)  
> z  
| (5, 6)
```

Unpacking a sequence with one element



Python shell

```
> x = [42]  # simple assignment
> x
| [42]
> x, = [42]  # unpacking, implicit parenthesis
> x
| 42
> (x,) = [42]  # unpacking
> x
| 42
> x, = [1, 2, 3]
| ValueError: too many values to unpack (expected 1)
```

Tuples vs lists: $a += b$

- Lists

Extends existing list, i.e., same as `a.extend(b)`

- Tuples

Must create a new tuple `a + b` and assign to `a` (since tuples are immutable)

Python shell

```
> (1, 2) + (3, 4)
| (1, 2, 3, 4)
> x = [1, 2]
> y = x
> y += [3, 4]
> y
| [1, 2, 3, 4]
> x
| [1, 2, 3, 4] 
> x = (1, 2)
> y = x
> y += (3, 4)
> y
| (1, 2, 3, 4)
> x
| (1, 2) 
```

More on += on lists

- Since `a += b` is the same as `a.extend(b)` we can also do

Python shell

```
> x = [1, 2, 3]
> x += [4, 5, 6]
> x += (7, 8, 9)
> x += range(10, 13)    # 10, 11, 12
> x += 'abc'           # 'a', 'b', 'c'
> print(x)
| [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 'a', 'b', 'c']
> x = [1, 2, 3] + (4, 5, 6)
| TypeError: can only concatenate list (not "tuple") to list
```

- For tuples += only accepts tuples

*variable assignment

- For a tuple of variable length a single **variable name* on the left side will be assigned a *list* of the remaining elements not matched by variables preceding/following *

- *Example*

`a, *b, c = t`

is equivalent to

`a = t[0]`

`b = t[1:-1]`

`c = t[-1]`

- There can be a single * in a left-hand-side tuple (but one new * in each nested tuple)

Python shell

```
> (a, *b, c, d) = (1, 2, 3, 4, 5, 6)
> b
| [2, 3, 4]
> (a, *b, c, d) = (1, 2, 3)
> b
| []
> (a, *b, c, d) = (1, 2)
| ValueError: not enough values to
  unpack (expected at least 3, got 2)
> v = ((1,2,3),4,5,6,(7,8,9,10))
> ((a, *b), *c, (d, *e)) = v
> b
| [2, 3]
> c
| [4, 5, 6]
> e
| [8, 9, 10]
> head, *tail = [1, 2, 3, 4]
> head
| 1
> tail
| [2, 3, 4]
```

Question – What is b ?

`(*a, (b, ), c) = ((1, 2), ((3, 4)), ((5, )), (6))`

- a) (1, 2)
- b) (3, 4)
-  c) 5
- d) (5,)
- e) (6)
- f) Don't know

Python shell

```
> (*a, (b, ), c) = ((1, 2), ((3, 4)), ((5, )), (6))
> a
| [(1, 2), (3, 4)]
> b
| 5
> c
| 6
```

* in list and tuple construction

- When constructing a list or tuple you can insert zero or more elements from another list/tuple/sequence by inserting **expression*
- There can be an arbitrary number of * expressions in a tuple or list construction

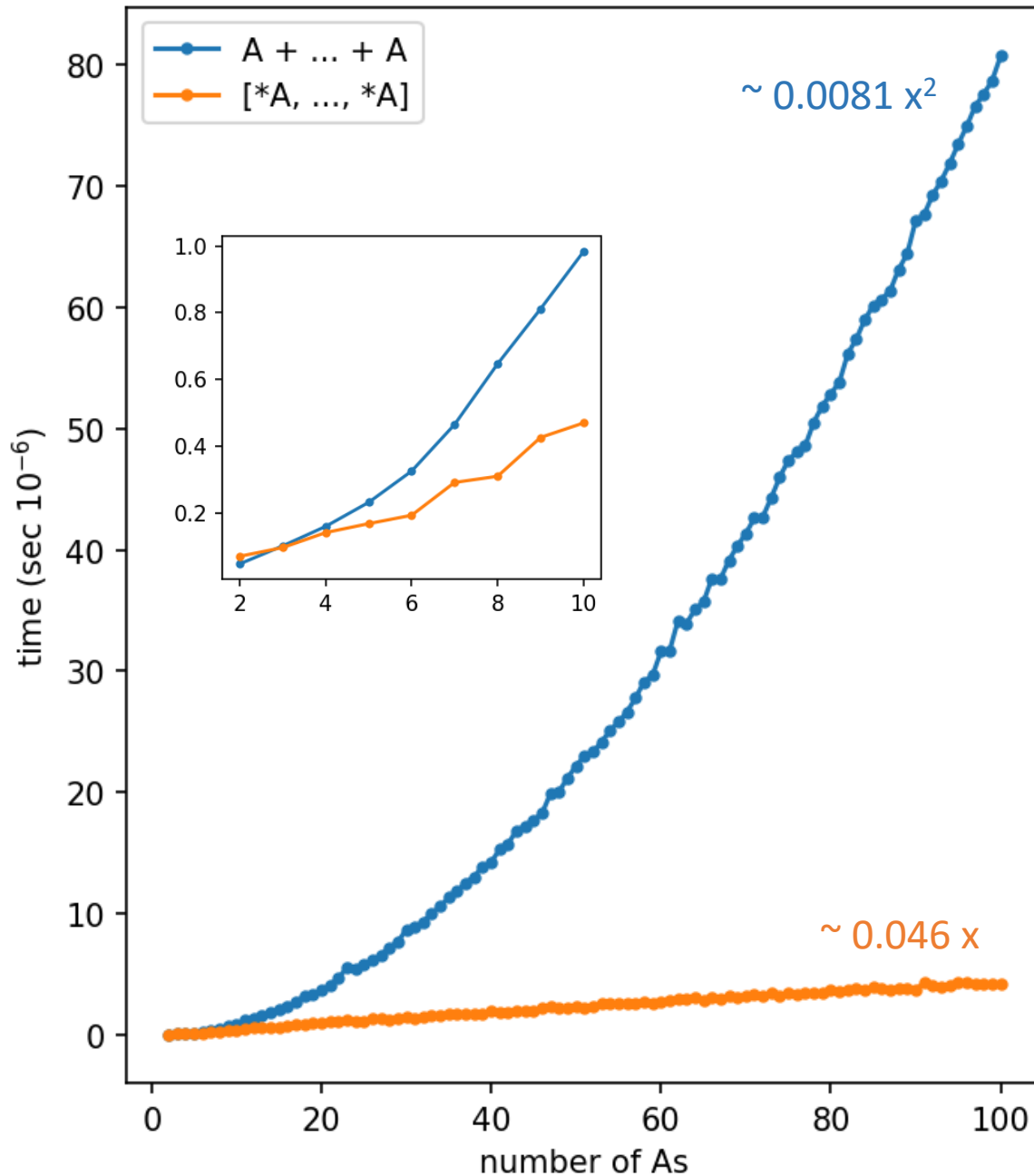
Python shell

```
> A = (1, 2, 3)
> B = ['B', 'C']
> L = [A, B, 4, 5]
> L
| [(1, 2, 3), ['B', 'C'], 4, 5]
> len(L)
| 4
> L = [*A, *B, 4, 5]
> L
| [1, 2, 3, 'B', 'C', 4, 5]
> len(L)
| 7
> (*A, *B, 4, 5)
| (1, 2, 3, 'B', 'C', 4, 5)
```

Python shell

```
> from timeit import timeit
> timeit('A + A + A + A + A + A + A + A', setup='A = [1,2,3,4,5,6,7,8,9,10]')
| 0.665172699955292 # repeated concatenation can be slow
> timeit('[*A, *A, *A, *A, *A, *A, *A, *A]', setup='A = [1,2,3,4,5,6,7,8,9,10]')
| 0.32599859999027103 # * notation can be faster for multiple concatenation
```





list_catenation.py

```
import matplotlib.pyplot as plt
from timeit import timeit

ns = range(2, 101)
P, S = [], []
for n in ns:
    setup = 'A = list(range(10))'
    plus = ' + '.join(['A'] * n)
    star = '[' + ', '.join(['*A'] * n) + ']'
    P.append(timeit(plus, setup=setup))
    S.append(timeit(star, setup=setup))

plt.plot(ns, P, '.-', label='A + ... + A')
plt.plot(ns, S, '.-', label='[*A, ..., *A]')
plt.legend()
plt.ylabel('time (sec  $10^{-6}$ )')
plt.xlabel('number of As')
plt.show()
```


List comprehension (cool stuff)

- Example:

```
[x*x for x in [1, 2, 3]]
```

returns

```
[1, 4, 9]
```

- General

```
[expression for variable in sequence]
```

returns a list, where *expression* is computed for each element in *sequence* assigned to *variable*

Python shell

```
> [2*x for x in [1,2,3]]  
| [2, 4, 6]  
> [2*x for x in (1,2,3)]  
| [2, 4, 6]  
> [2*x for x in range(10,15)]  
| [20, 22, 24, 26, 28]  
> [2*x for x in 'abc']  
| ['aa', 'bb', 'cc']  
> [(None, None) for _ in range(2)]  
| [(None, None), (None, None)]
```

List comprehension (it's just syntactic sugar...)

Python shell

```
> [x * 2 for x in [1, 2, 3]]  
| [2, 4, 6]  
  
> L = []  
> for x in [1, 2, 3]:  
>     L.append(x * 2)  
> L  
| [2, 4, 6]
```

List comprehension (more cool stuff)

- Similarly, to the left-hand-side in assignments, the variable part can be a (nested) tuple of variables for unpacking elements:

[expression for tuple of variables in sequence]

Python shell

```
> points = [(3, 4), (2, 5), (4, 7)]  
> [(x, y, x*y) for (x, y) in points]  
| [(3, 4, 12), (2, 5, 10), (4, 7, 28)]  
> [(x, y, x*y) for x, y in points]  
| [(3, 4, 12), (2, 5, 10), (4, 7, 28)]  
> [x, y, x*y for (x, y) in points]  
| SyntaxError: invalid syntax
```



parenthesis required for
the constructed tuples

List comprehension – for-if and multiple for

- List comprehensions can have nested for-loops

`[expression for v1 in s1 for v2 in s2 for v3 in s3]`

- Can select a subset of the elements by adding an if-condition

`[expression for v1 in s1 if condition]`

- and be combined...

Python shell

```
> [(x, y) for x in range(1, 3) for y in range(4, 6)]  
| [(1, 4), (1, 5), (2, 4), (2, 5)]
```

```
> [x for x in (1, 2) for x in (4, 5)]  
| [4, 5, 4, 5]
```



```
> [x for x in range(1, 101) if x % 7 == 1 and x % 5 == 2]  
| [22, 57, 92]
```

```
> [(x, y, x*y) for x in range(1, 11) if 6 <= x <= 7 for y in range(x, 11) if 6 <= y <= 7 and not x == y]  
| [(6, 7, 42)]
```

Question – What will print the same?

```
points = [(3,7), (4,10), (12,3), (9,11), (7,5)]  
print([(x, y) for x, y in points if x < y])
```

- a) `print([x, y for x, y in points if x < y])`
- b) `print([(x, y) for p in points if p[0] < p[1]])`
-  c) `print([p for p in points if p[0] < p[1]])`
- d) `print([[x, y] for x, y in points if x < y])`
- e) Don't know

List comprehension – space usage long lists ⚠

Bad: creates new tuples
(64 bytes per point)

Good: looks nice

`list_comprehension_space_usage.py`

3.7 MB

```
input('press enter')
```

1305.6 MB

```
points = [(x, x + 1) for x in range(10_000_000)] # dummy points
```

```
input('press enter')
```

1997.3 MB

```
selected = [(x, y) for (x, y) in points if y == x + 1]
```

```
input('press enter')
```

1387.3 MB

```
selected = [point for point in points if point[1] == point[0] + 1]
```

```
input('press enter')
```

Good: refers to existing tuple
(8 bytes per point)

Bad: looks ugly (hard to read)

- Memory usage according to “Task Manager” in Windows
- Unpacking makes code more readable, but creating new tuples requires additional space...

Without comprehension (1 line becomes 5 lines)

Good: looks nice

3.8 MB

1305.6 MB

1383.0 MB

`list_unpacking_space_usage.py`

```
input('press enter')
points = [(x, x + 1) for x in range(10_000_000)]
input('press enter')
selected = []
for point in points:
    x, y = point # unpack tuple
    if y == x + 1:
        selected.append(point) # append original tuple
input('press enter')
```

Good: refers to existing tuple

any, all

- `any(L)` checks if at least one element in the sequence `L` is true (list, tuple, ranges, sequence, strings, ...)

```
any([False, True, False])
```

- `all(L)` checks if all elements in the sequence `L` are true

```
all([False, False, True])
```

- `any` and `all` return `True` or `False`

Python shell

```
> any((False, True, False))
| True
> any([False, False, False])
| False
> any([])
| False
> all([False, False, True])
| False
> all((True, True, True))
| True
> all(())
| True
> L = (7, 42, 13)
> any([x == 42 for x in L])
| True
> all([x == 42 for x in L])
| False
```


Example – computing primes

Python shell

```
> [x for x in range(2, 50) if all([x % f for f in range(2, x)])]  
| [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]  
  
> [10 % f for f in range(2, 10)]  
| [0, 1, 2, 0, 4, 3, 2, 1]  
> all([10 % f for f in range(2, 10)]) # == 0 is considered False  
| False  
> [13 % f for f in range(2, 13)]  
| [1, 1, 1, 3, 1, 6, 5, 4, 3, 2, 1]  
> all([13 % f for f in range(2, 13)])  
| True
```

enumerate

`list(enumerate(L))`

returns

`[(0, L[0]), (1, L[1]), ..., (len(L) - 1, L[-1])]`

Python shell

```
> points = [(1, 2), (3, 4), (5, 6)]
> [(idx, x * y) for idx, (x, y) in enumerate(points)]
| [(0, 2), (1, 12), (2, 30)]
> L = ('a', 'b', 'c')
> list(enumerate(L))
| [(0, 'a'), (1, 'b'), (2, 'c')]
> L_ = []
> for idx in range(len(L)):
>     L_.append((idx, L[idx]))
> print(L_)
| [(0, 'a'), (1, 'b'), (2, 'c')]
> list(enumerate(['a', 'b', 'c'], start=7))
| [(7, 'a'), (8, 'b'), (9, 'c')]
```

zip

`list(zip(L1, L2, ..., Lk)) = [ (L1[0], L2[0], ..., Lk[0]) , ..., (L1[n-1], L2[n-1], ..., Lk[n-1]) ]`
where $n = \min(\text{len}(L_1), \text{len}(L_2), \dots, \text{len}(L_k))$

- Example (“matrix transpose”):

```
list(zip([1, 2, 3],  
        [4, 5, 6],  
        [7, 8, 9]))
```

returns

```
[(1, 4, 7),  
 (2, 5, 8),  
 (3, 6, 9)]
```

Python shell

```
> x = [1, 2, 3]
> y = [4, 5, 6]
| zip(x, y)
> <zip at 0xb02b530>
> points = list(zip(x, y))
> print(points)
| [(1, 4), (2, 5), (3, 6)]
```

Python shell

```
> first = ['Donald', 'Mickey', 'Scrooge']  
> last = ['Duck', 'Mouse', 'McDuck']  
  
> for i, (a, b) in enumerate(zip(first, last), start=1):  
>     print(i, a, b)  
  
| 1 Donald Duck  
| 2 Mickey Mouse  
| 3 Scrooge McDuck
```

(Simple) functions

- You can define your own functions using:

```
def function-name (var1, ..., vark):  
    body code
```

- If the body code executes

```
return expression
```

the result of *expression* will be returned by the function. If expression is omitted or the body code terminates without performing `return`, then `None` is returned

- When *calling* a function *name* (*value*₁, ..., *value*_{*k*}) body code is executed with *var*_{*i*}=*value*_{*i*},

Python shell

```
> def sum3(x, y, z):  
    return x + y + z  
  
> sum3(1, 2, 3)  
| 6  
> sum3(5, 7, 9)  
| 21  
  
> def powers(L, power):  
    P = [x**power for x in L]  
    return P  
  
> powers([2, 3, 4], 3)  
| [8, 27, 64]
```

Question – What tuple is printed ?

```
def even(x):  
    if x % 2 == 0:  
        return True  
    else:  
        return False  
  
print (even(7), even(6))
```

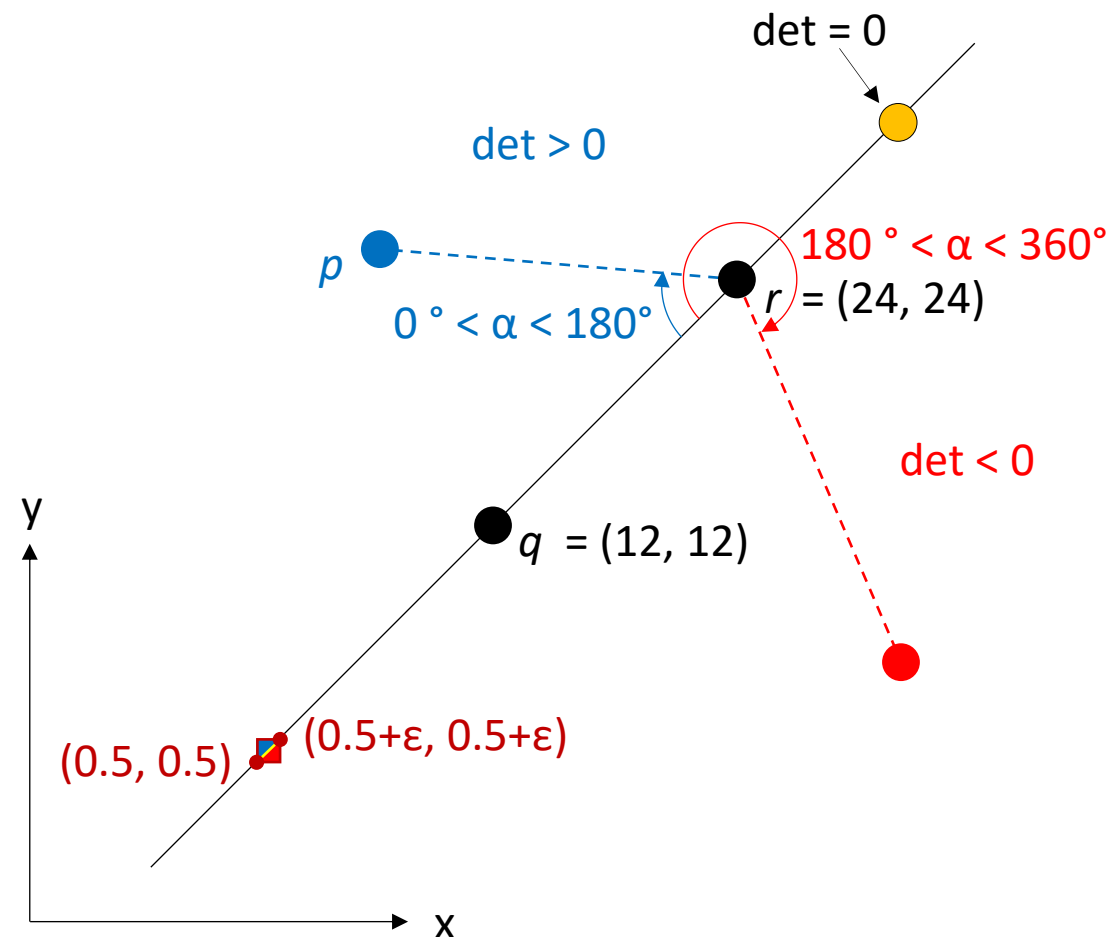


- a) (False, False)
- b) (False, True)
- c) (True, False)
- d) (True, True)
- e) Don't know

Geometric orientation test

Purpose of example

- illustrate tuples
- list comprehension
- matplotlib.pyplot
- floats are strange



$$\det = \begin{vmatrix} 1 & q_x & q_y \\ 1 & r_x & r_y \\ 1 & p_x & p_y \end{vmatrix} = \underbrace{r_x p_y - p_x r_y - q_x p_y + p_x q_y + q_x r_y - r_x q_y}_{=0}$$

6 ! = 720 different orders to add



sign-plot.py

```
import matplotlib.pyplot as plt

N = 256
delta = 1 / 2**54
q = (12, 12)
r = (24, 24)
P = [] # points (i, j, det)

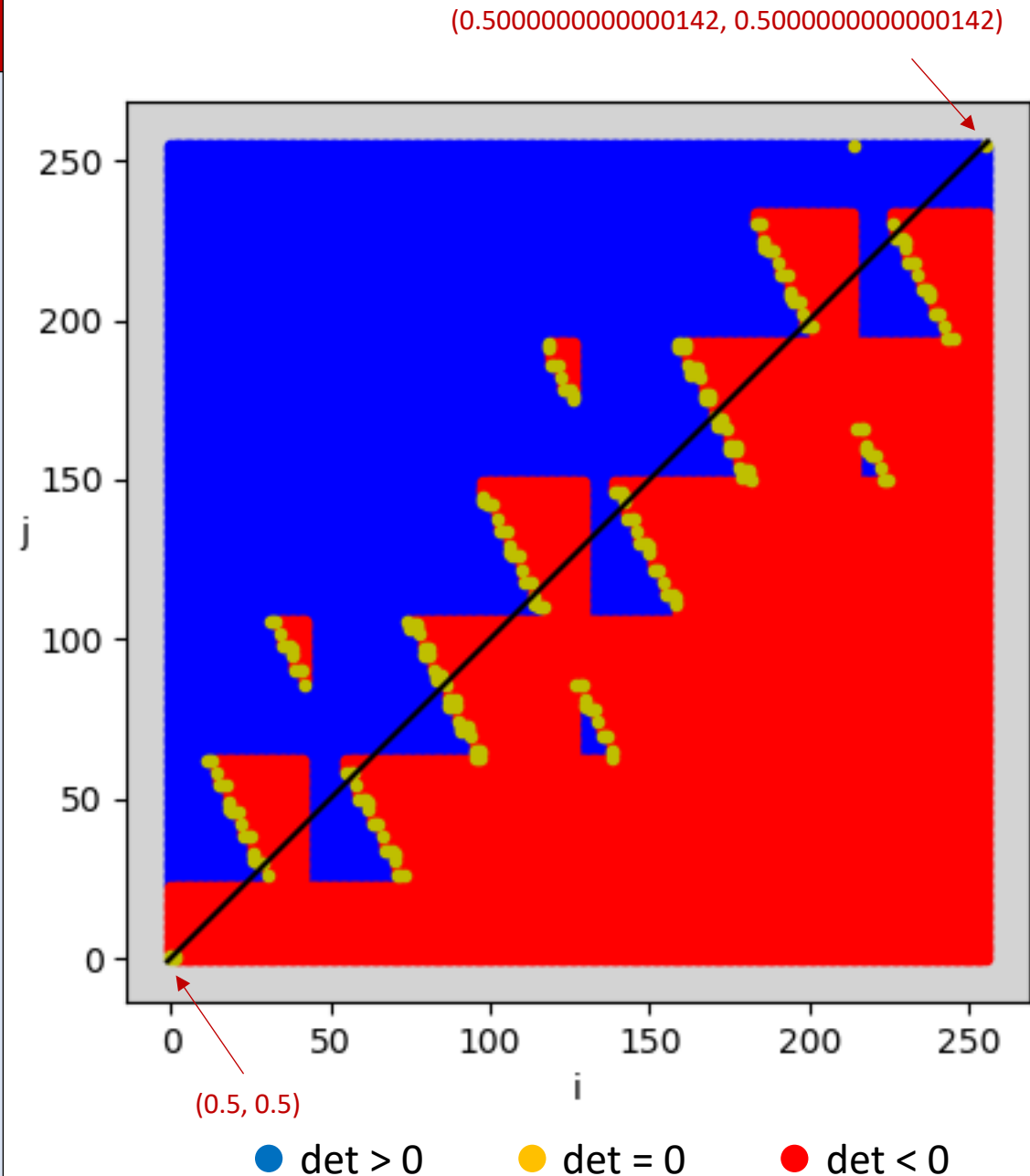
for i in range(N):
    for j in range(N):
        p = (1/2 + i * delta, 1/2 + j * delta)
        det = (q[0]*r[1] + r[0]*p[1] + p[0]*q[1]
              - r[0]*q[1] - p[0]*r[1] - q[0]*p[1])
        P.append((i, j, det))

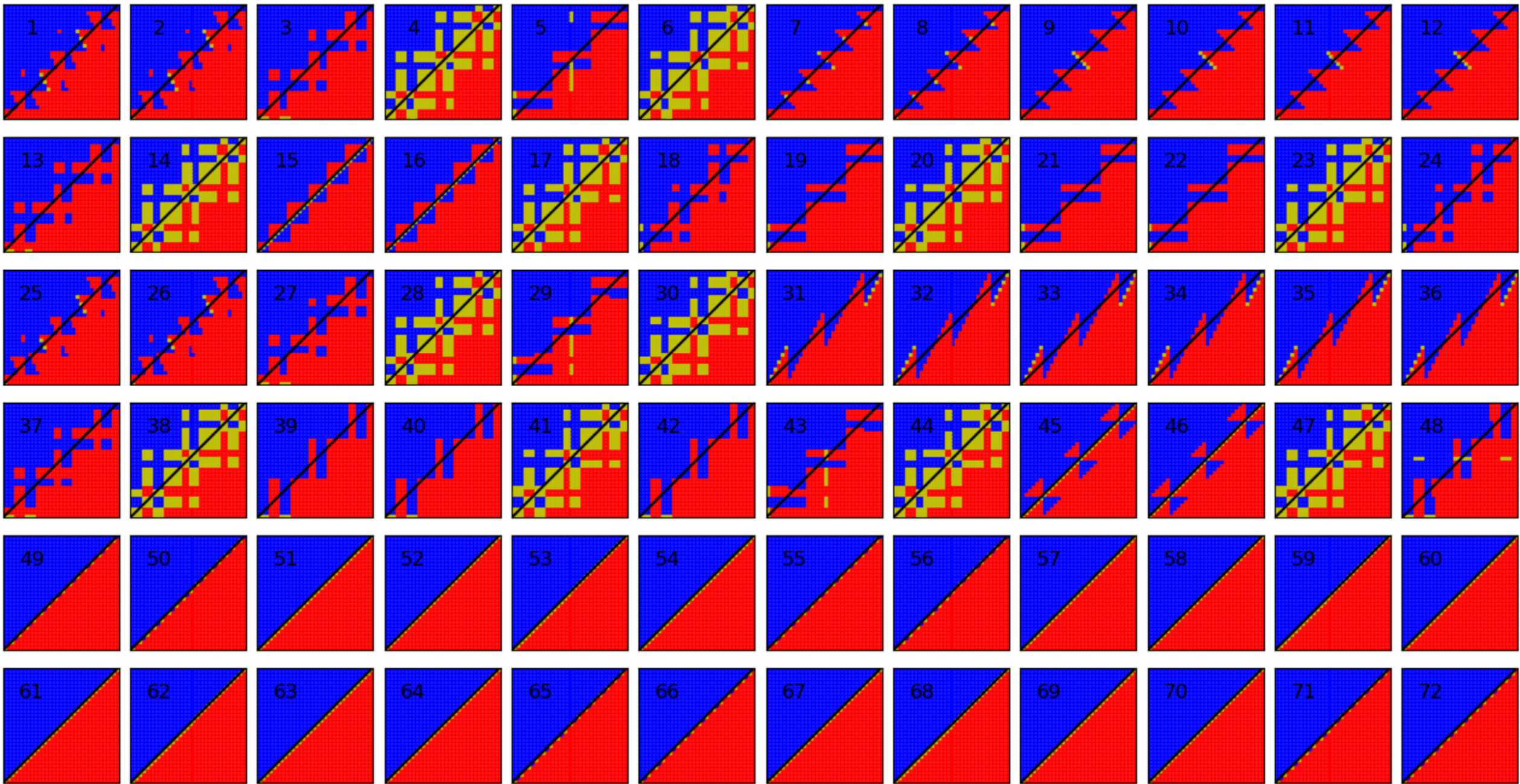
pos = [(i, j) for i, j, det in P if det > 0]
neg = [(i, j) for i, j, det in P if det < 0]
zero = [(i, j) for i, j, det in P if det == 0]

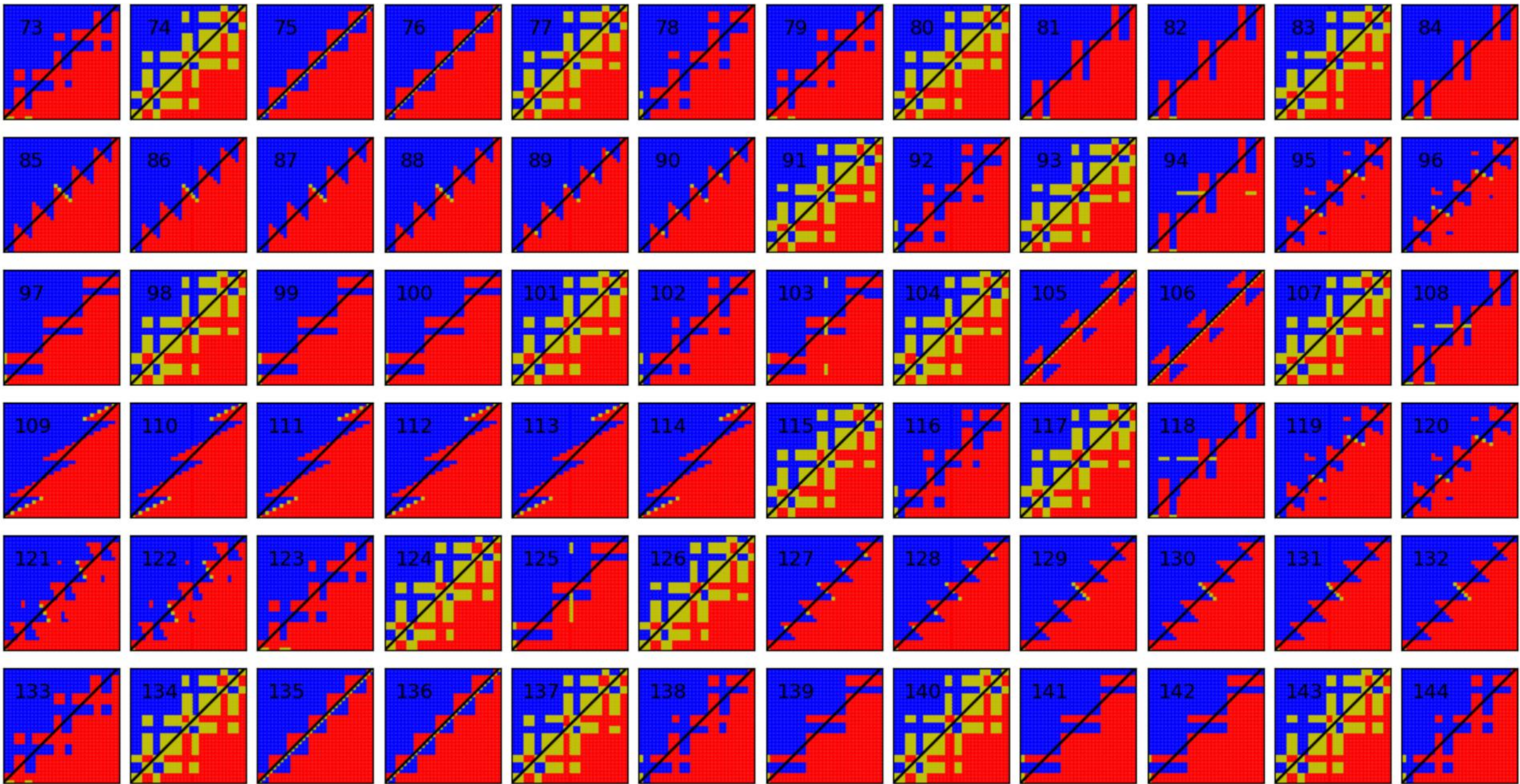
plt.subplot(facecolor='lightgrey', aspect='equal')
plt.xlabel('i')
plt.ylabel('j', rotation=0)

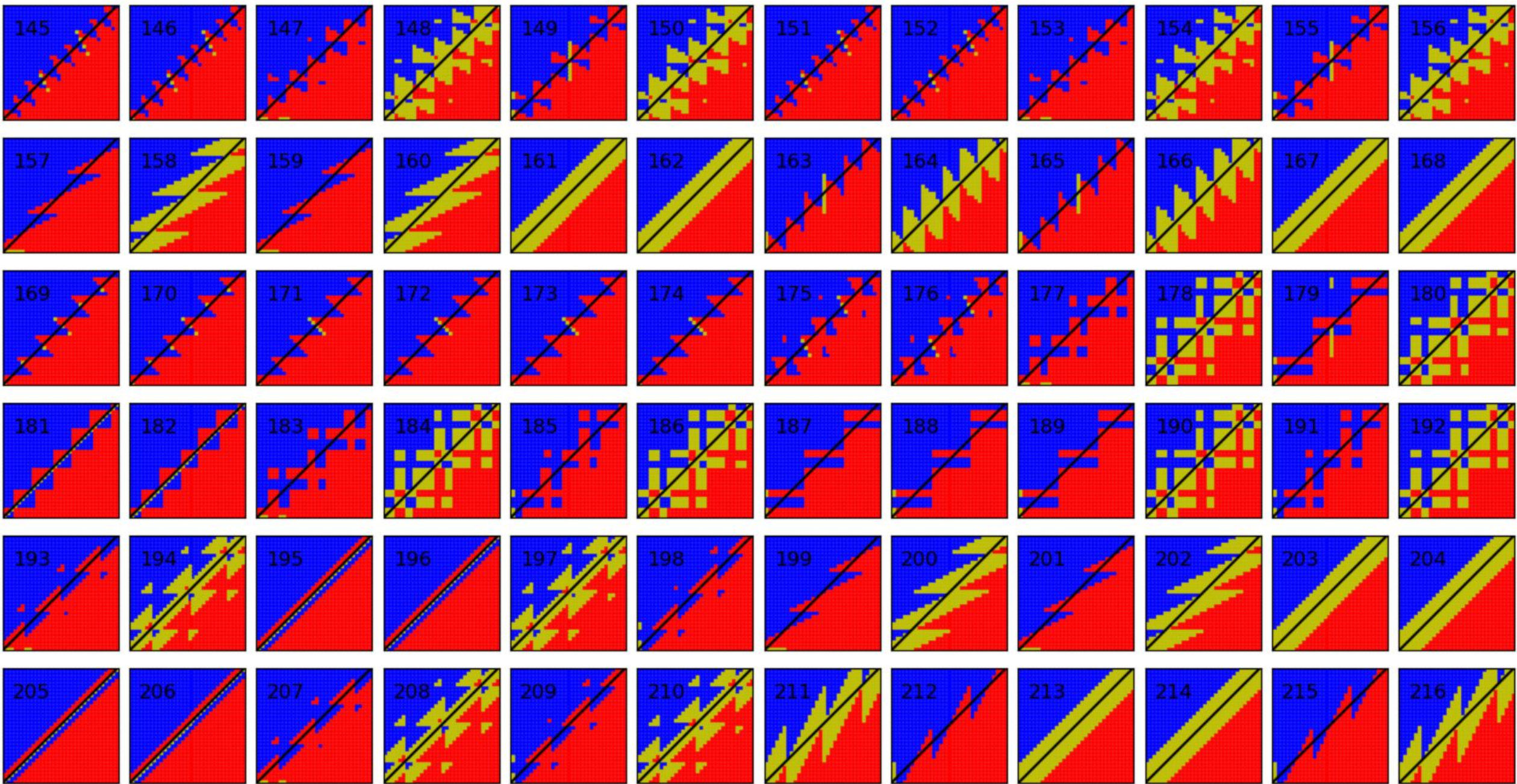
for points, color in [(pos, 'b'), (neg, 'r'), (zero, 'y')]:
    X = [i for i, j in points]
    Y = [j for i, j in points]
    plt.plot(X, Y, color + '.')

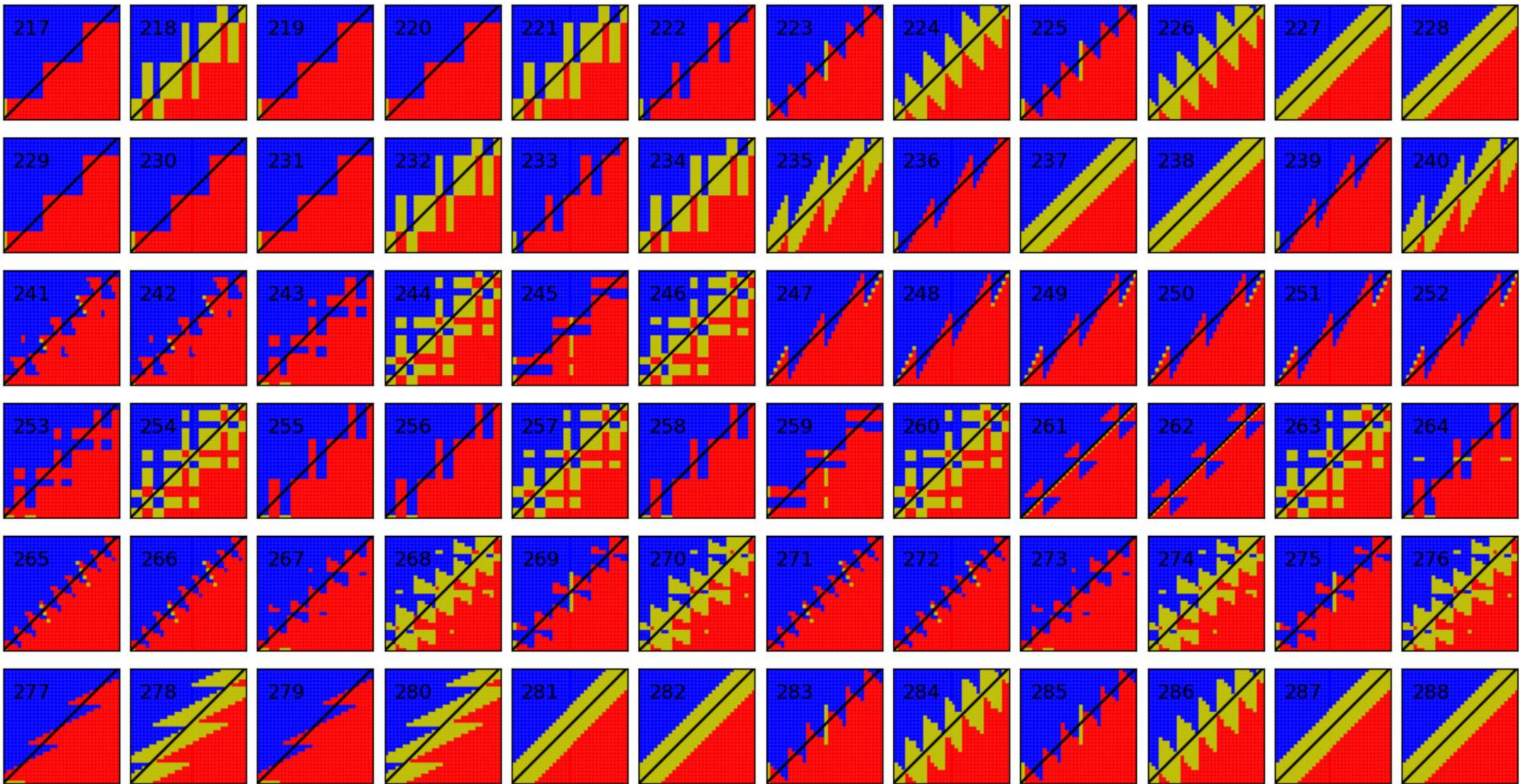
plt.plot([-1, N], [-1, N], 'k-')
plt.show()
```

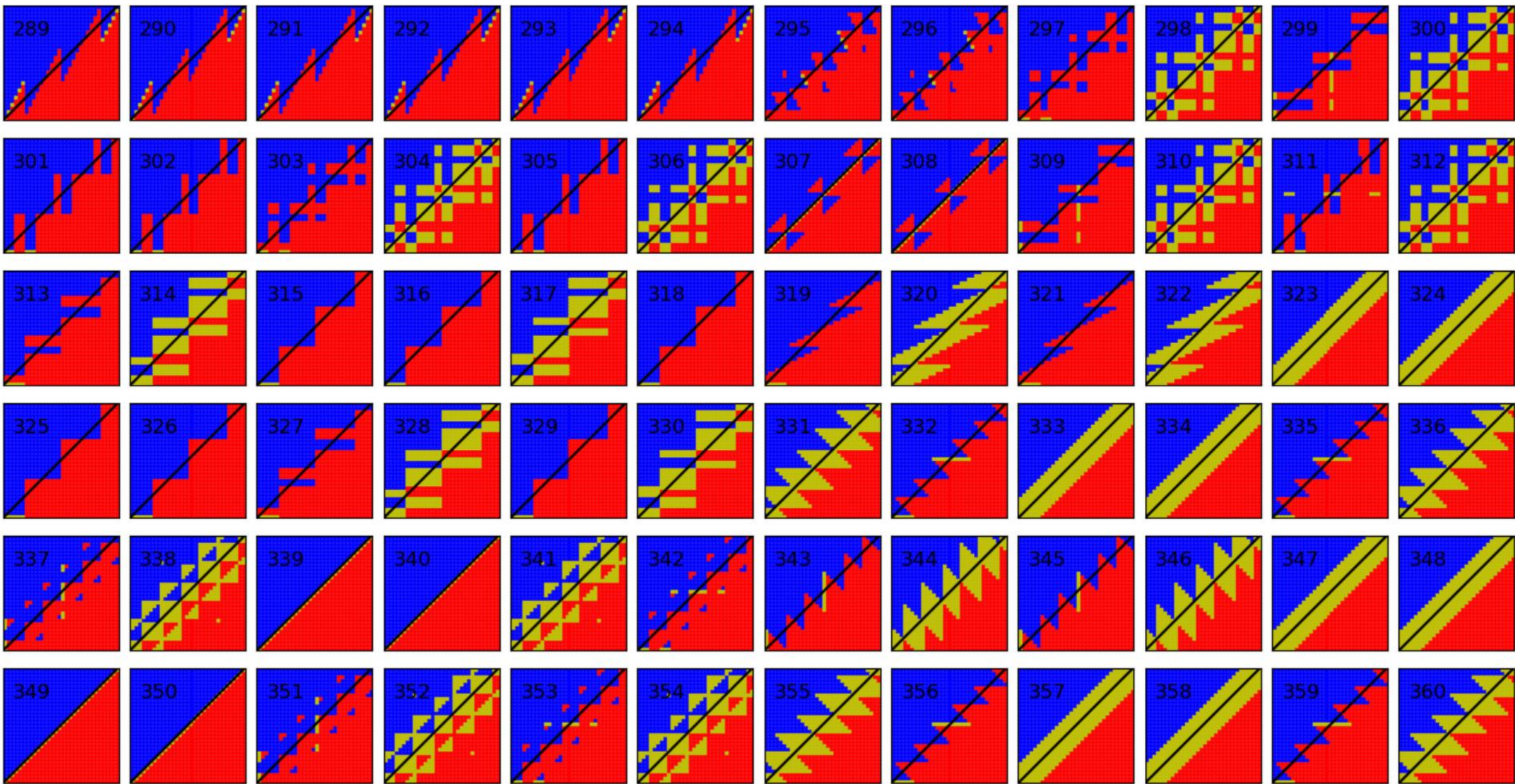


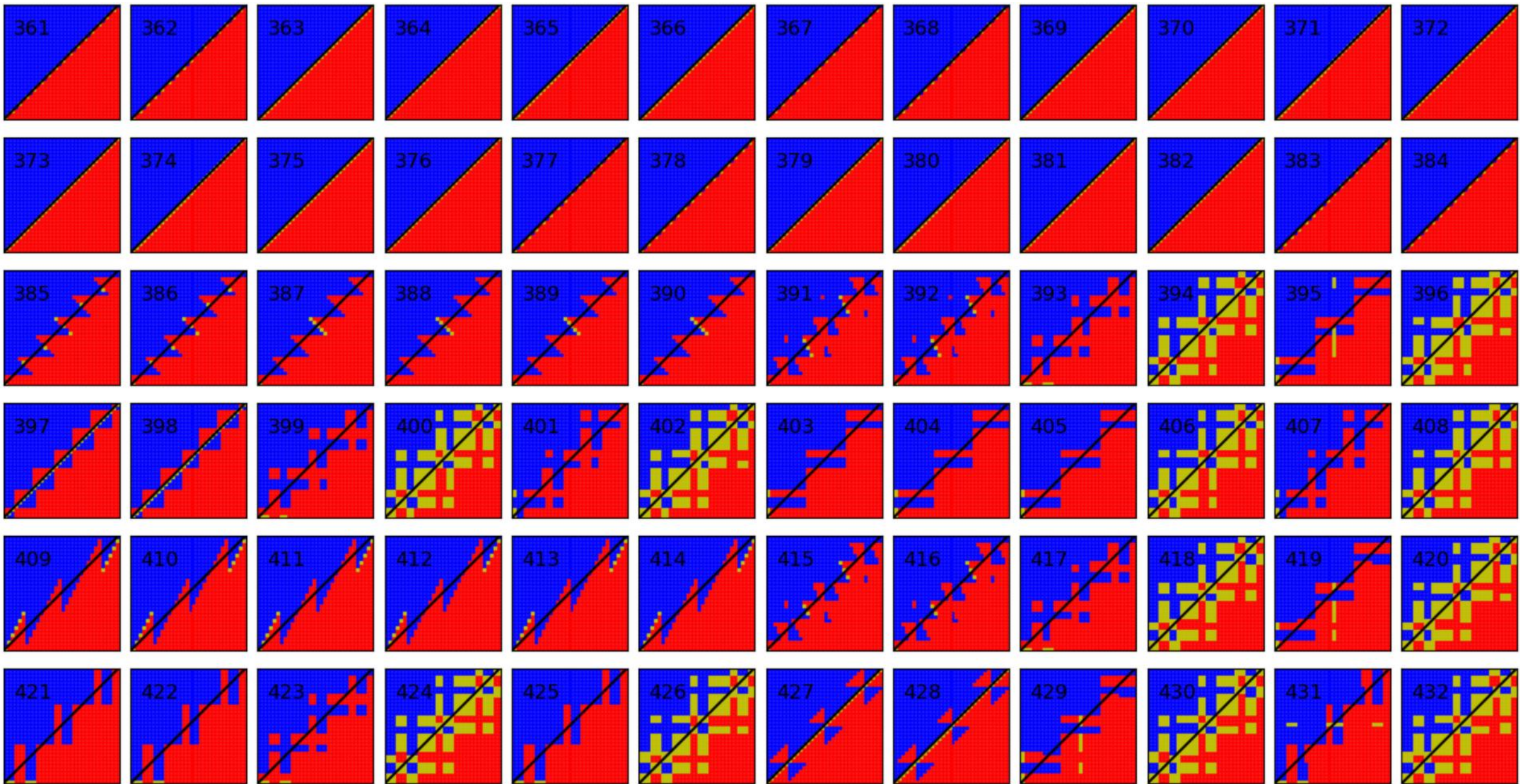


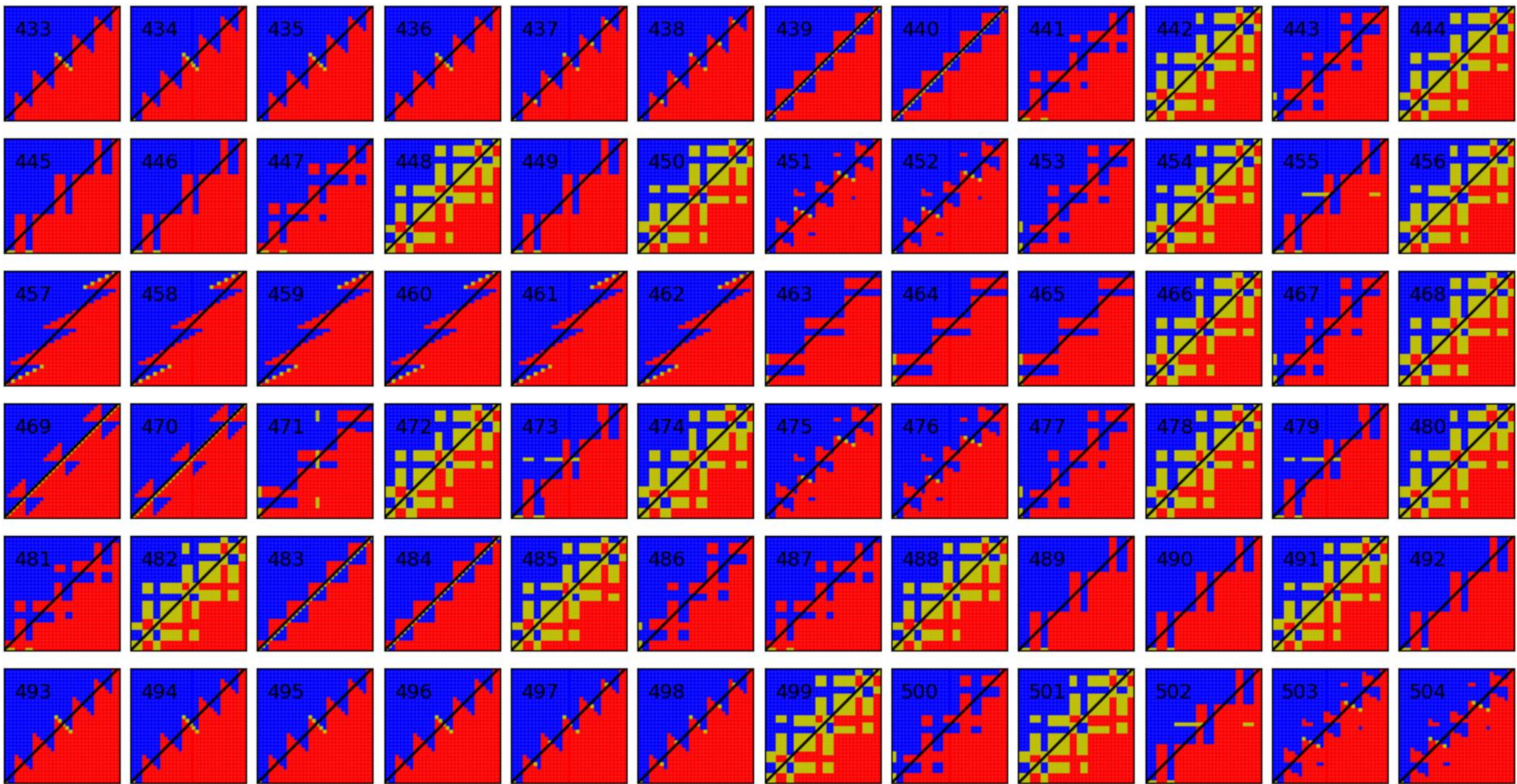


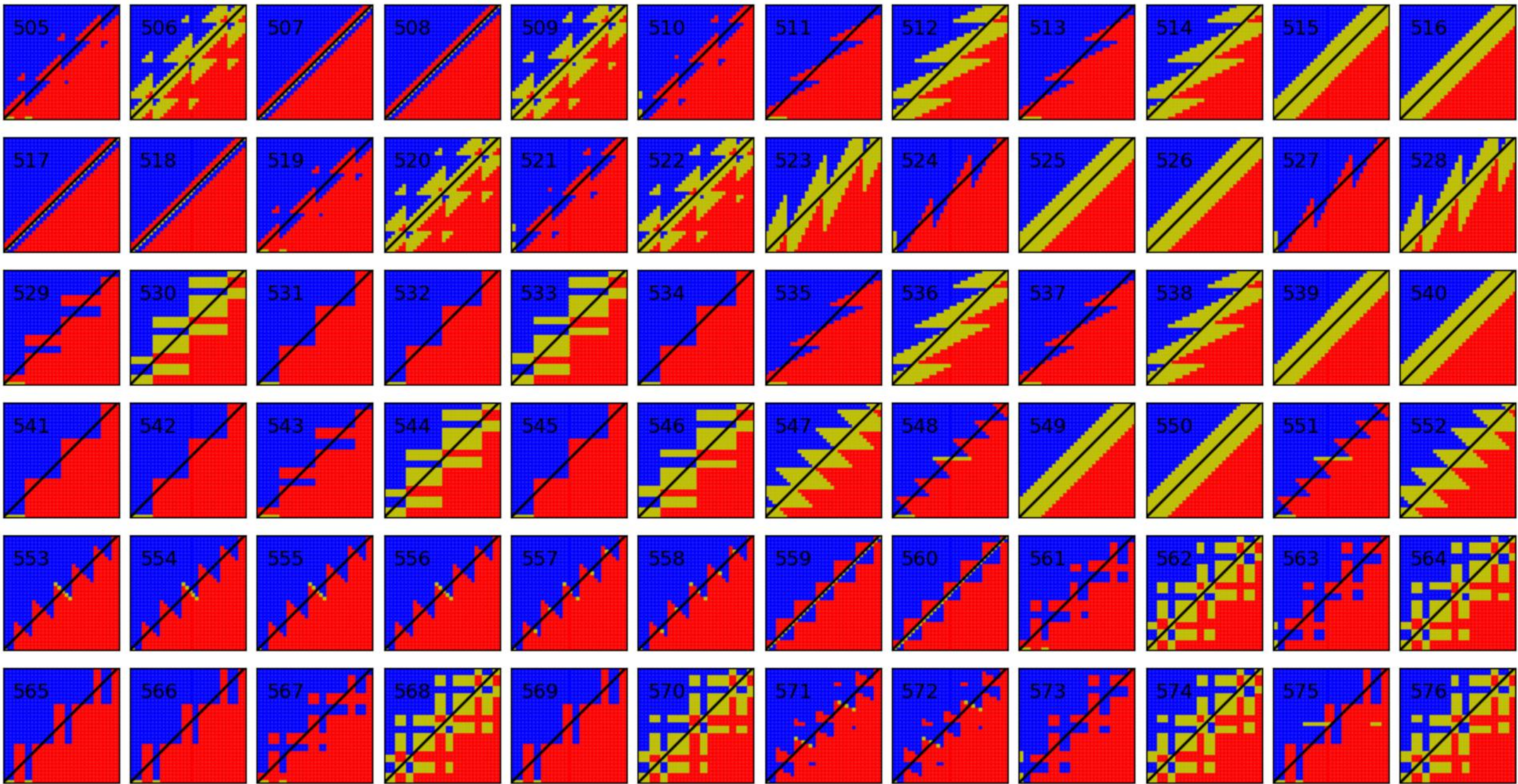


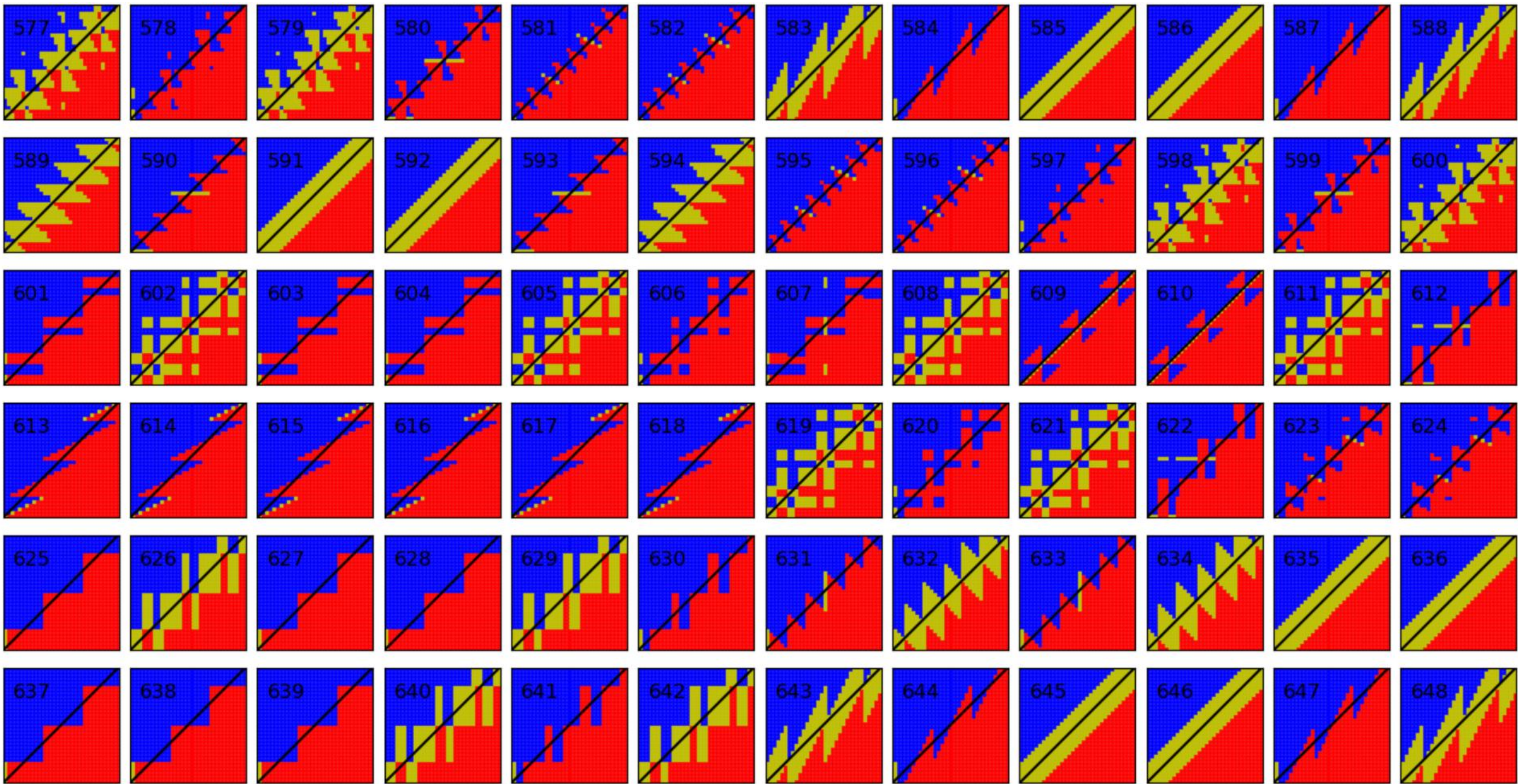


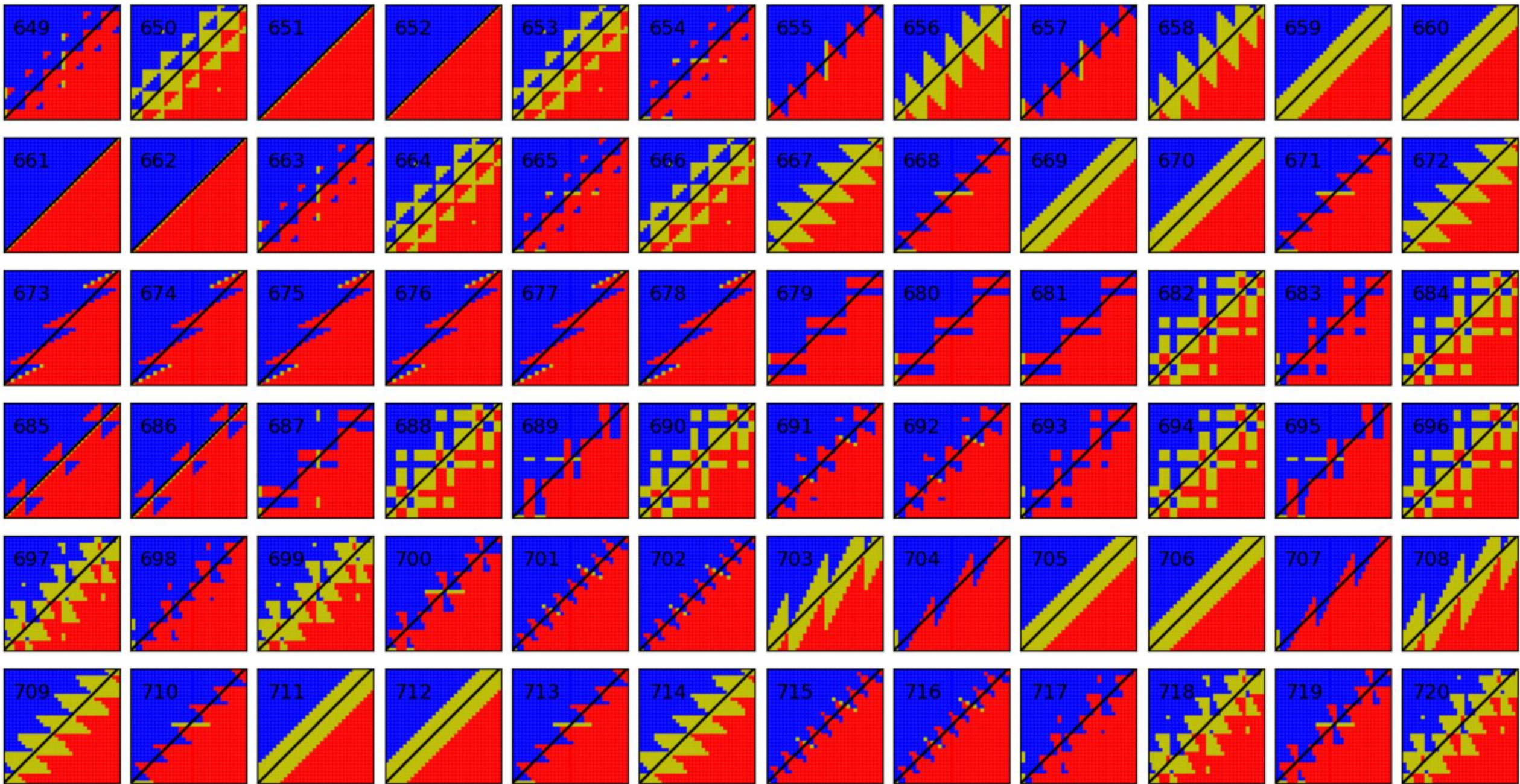












Dictionaries and Sets

- dict
- set
- frozenset
- set/dict comprehensions

Dictionaries (type dict)

$\{ key_1: value_1, \dots, key_k: value_k \}$

- Stores a mutable set of (key, value) pairs, denoted *items*, with distinct keys, i.e. *maps* keys to values
- Constructing empty dictionary: `dict()` or `{}`
- `dict[key]` lookup for key in dictionary, and returns associated value. Key must be present, otherwise a `KeyError` is raised
- `dict[key] = value` assigns value to *key*, overriding existing value if present

key	value
'a'	7
'foo'	'42nd'
5	29
'5'	44
5.5	False
False	True
(3, 4)	'abc'



distinct keys,
i.e. not "=="

Dictionaries (type dict)

Python shell

```
> d = {'a': 42, 'b': 57}
> d
| {'a': 42, 'b': 57}

> d.keys()
| dict_keys(['a', 'b'])

> list(d.keys())
| ['a', 'b']

> d.items()
| dict_items([('a', 42), ('b', 57)])

> list(d.items())
| [('a', 42), ('b', 57)]
```

```
> for key in d:
    print(key)
| a
| b

> for key, val in d.items():
    print('Key', key, 'has value', val)
| Key a has value 42
| Key b has value 57

> {5: 'a', 5.0: 'b'}
| {5: 'b'}
```



Python shell

```
> surname = dict(zip(['Donald', 'Mickey', 'Scrooge'], ['Duck', 'Mouse', 'McDuck']))
> surname['Mickey']
| 'Mouse'
```

Dictionaries (type dict)

Python shell

```
> gradings = [('A', 7), ('B', 4), ('A', 12), ('C', 10), ('A', 7)]
> grades = {} # empty dictionary
> for student, grade in gradings:
    if student not in grades: # is key in dictionary
        grades[student] = []
    grades[student].append(grade)
> grades
| {'A': [7, 12, 7], 'B': [4], 'C': [10]}
> print(grades['A'])
| [7, 12, 7]
> print(grades['E']) # can only lookup keys in dictionary
| KeyError: 'E'
> print(grades.get('E')) # .get returns None if key not in dictionary
| None
> print(grades.get('E', [])) # change default return value
| []
> print(grades.get('A', []))
| [7, 12, 7]
```

Dictionary initialization

Python shell

```
> d1 = {'A': 7, 'B': 42}
> d2 = dict([('A', 7), ('B', 42)]) # list of (key, value) pairs
> d3 = dict(A=7, B=42)             # keyword arguments to constructor
> d1 == d2 == d3
| True
```

- **Note:** *keyword initialization* only works if keys are strings which are valid keyword arguments to a function – but saves writing a lot of quotes

Python shell

```
> d1 = dict(A=1, B=2)
> d2 = dict(B=3, C=4)
> d1 | d2                               # merge dictionaries
| {'A': 1, 'B': 3, 'C': 4}             # rightmost value for 'B' wins
> {**d1, **d2, 'D': 5}                 # ** inserts dictionary content
| {'A': 1, 'B': 3, 'C': 4, 'D': 5}    # rightmost value for 'B' wins
> d1 |= d2                             # same as d1.update(d2)
> d1
| {'A': 1, 'B': 3, 'C': 4}
```



| and |=

new in
Python 3.9
PEP 584

Dictionary operation	Description
<code>len(d)</code>	Items in dictionary
<code>d[key]</code>	Lookup key
<code>d[key] = value</code>	Update value of key
<code>del d[key]</code>	Delete an existing key
<code>key in d</code>	Key membership
<code>key not in d</code>	Key non-membership
<code>clear()</code>	Remove all items
<code>copy()</code>	Shallow copy
<code>get(key), get(key, default)</code>	<code>d[key]</code> if key in dictionary, otherwise <code>None</code> or default
<code>items()</code>	<i>View</i> of the dictionaries items
<code>keys()</code>	<i>View</i> of the dictionaries keys
<code>values()</code>	<i>View</i> of the dictionaries values
<code>pop(key)</code>	Remove key and return previous value
<code>popitem()</code>	Remove and return an arbitrary item
<code>update()</code>	Update key/value pairs from another dictionary

Tuples as dictionary keys

- A tuple can be used as a dictionary key, and parenthesis can be omitted

Python shell -

```
> d = {('a', 1): 7, ('b', 2): 42}
> d[('b', 2)]
| 42
> d['b', 2] # same as above, parenthesis omitted
| 42
> T = [[None, None], [42, None]] # 2D table as lists-of-lists
> T[1][0]
| 42
> T[1, 0] # wrong, T is a list (of lists)
| TypeError: list indices must be integers or slices, not tuple
> T = {(1, 0): 42} # 2D table as dictionary
> T[1, 0] # dictionary lookup with tuple (1, 0) as key
| 42
> T[1][0] # wrong, T has only one key = the tuple (1, 0)
| KeyError: 1
```

Order returned by `list(d.keys())` ?

The Python Standard Library Mapping Types — dict

“Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.” (since Python 3.7)

docs.python.org/3/library/stdtypes.html

Python shell

```
> d = {'d': 1, 'c': 2, 'b': 3, 'a': 4}
> d['x'] = 5      # new key at end
> d['c'] = 6      # overwrite value
> del d['b']      # remove key 'b'
> d['b'] = 7      # reinsert key 'b' at end
> d
| {'d': 1, 'c': 6, 'a': 4, 'x': 5, 'b': 7}
```



Raymond Hettinger @ Twitter

See also Raymond's talk @ PyCon 2017
Modern Python Dictionaries
A confluence of a dozen great ideas

Dictionary comprehension

- Similarly to creating a list using list comprehension, one can create a set of key-value pairs:

{key : value for variable in list}

Python shell

```
> names = ['Mickey', 'Donald', 'Scrooge']
> list(enumerate(names, start=1))
| [(1, 'Mickey'), (2, 'Donald'), (3, 'Scrooge')]
> dict(enumerate(names, start=1)) # construct dict from pairs
| {1: 'Mickey', 2: 'Donald', 3: 'Scrooge'}
> {name: idx for idx, name in enumerate(names, start=1)}
| {'Mickey': 1, 'Donald': 2, 'Scrooge': 3}
```

Sets (set and frozenset)

$\{value_1, \dots, value_k\}$

- Values of type `set` represent mutable sets, where "==" elements only appear once
- Do **not** support: indexing, slicing
- `frozenset` is an immutable version of `set`

Python shell	
> S = {2, 5, 'a', 'c'}	> S ^ T
> T = {3, 4, 5, 'c'}	{2, 3, 4, 'a'}
> S T	> S - T
{2, 3, 4, 5, 'a', 'c'}	{2, 'a'}
> S & T	> {4, 5, 5.0, 5.1}
{5, 'c'}	{4, 5, 5.1}



Operation	Description
$S \mid T$	Set union
$S \& T$	Set intersection
$S - T$	Set difference
$S \wedge T$	Symmetric difference
<code>set()</code>	Empty set (<code>{}</code> = empty dictionary)
<code>set(L)</code>	Create set from sequence
<code>x in S</code>	Membership
<code>x not in S</code>	Non-membership
<code>S.isdisjoint(T)</code>	Disjoint sets
$S \leq T$	Subset
$S < T$	Proper subset
$S \geq T$	Superset
$S > T$	Proper superset
<code>len(S)</code>	Size of S
<code>S.add(x)</code>	Add x to S (not frozenset)

<https://docs.python.org/3/tutorial/datastructures.html#sets>

<https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset>

Question – What value has the expression ?

```
sorted( { 5, 5.5, 5.0, '5' } )
```

a) { '5', 5, 5.0, 5.5 }

b) { 5, 5.5 }

c) ['5', 5.0, 5.5]

d) ['5', 5, 5.5]



e) TypeError

f) Don't know

Sets of (frozen) sets

- Sets are mutable, i.e. cannot be used as dictionary keys or elements in sets
- Frozen sets can



Python shell

```
> S = {'a', {'a', 'b'}, {'a', 'c'}}
| TypeError: unhashable type: 'set'
> S = {frozenset({'a'}), frozenset({'a', 'b'}), frozenset({'a', 'c'})}
> frozenset({'a', 'b'}) in S
| True
> {'a', 'b'} in S # automatically converts unhashable set to frozenset
| True
> {'a', 'b'} == frozenset(['a', 'b']) # frozenset from list
| True
> D = {frozenset({'a', 'b'}): 42} # dictionary
> frozenset({'a', 'b'}) in D
| True
> {'a', 'b'} in D # dictionaries are not that friendly as sets
| TypeError: unhashable type: 'set'
```

Set comprehension

- Similarly to creating a list using list comprehension, one can create a set of values (also using nested for- and if-statements):

{value for variable in list}

- A value occurring multiple times as *value* will only be included once

primes_set.py

```
n = 101
not_primes = {m for f in range(2, n) for m in range(2 * f, n, f)}
primes = set(range(2, n)) - not_primes
```

Python shell

```
> L = ['a', 'b', 'c']
> {(x,(y,z)) for x in L for y in L for z in L if x != y and y != z and z != x}
| {('a',('b','c')),('a',('c','b')),('b',('a','c')),...,('c',('b','a'))}
> L = {'a', 'b', 'c'}
> {(x,(y,z)) for x in L for y in L - {x} for z in L - {x, y}}
| {('c',('b','a')),('c',('a','b')),('a',('c','b')),...,('b',('a','c'))}
```

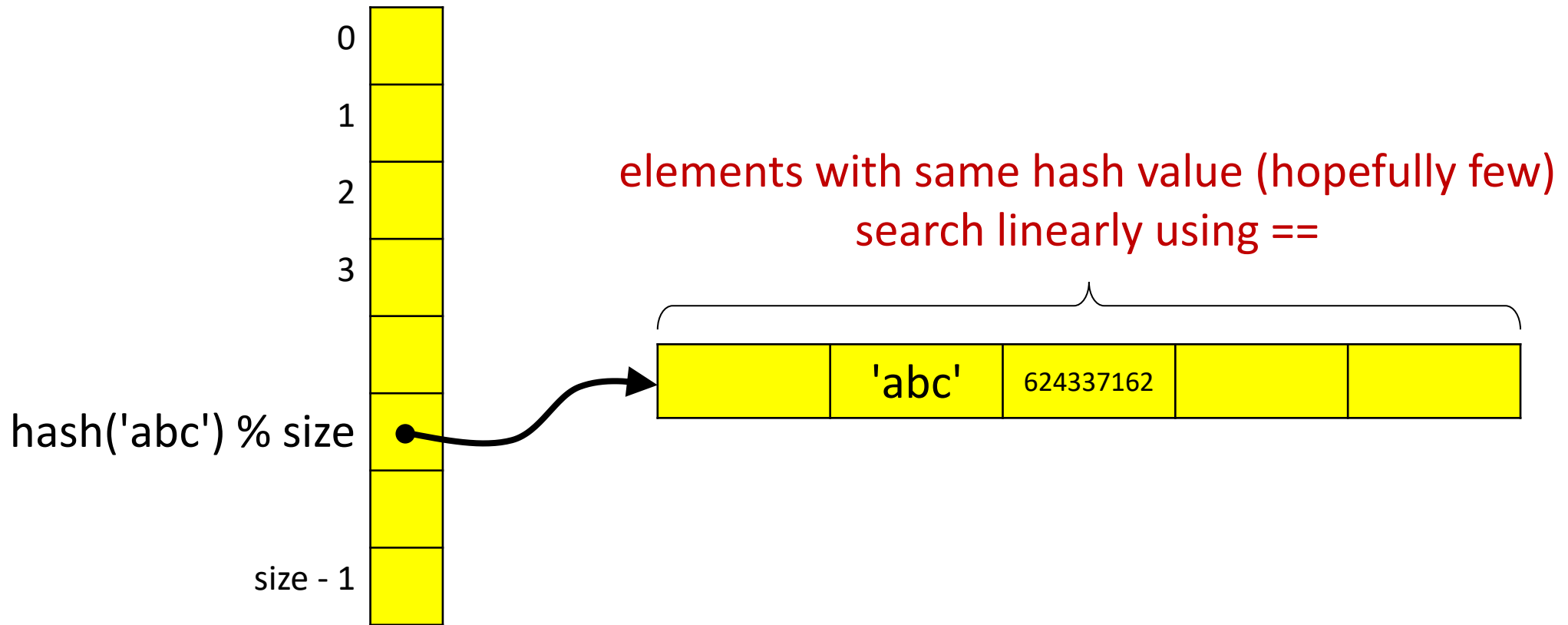
Hash, equality, and immutability

- Keys for dictionaries and sets must be *hashable*, i.e. have a `__hash__()` method returning an integer that does not change over their lifetime and an `__eq__()` method to check for equality with “==”

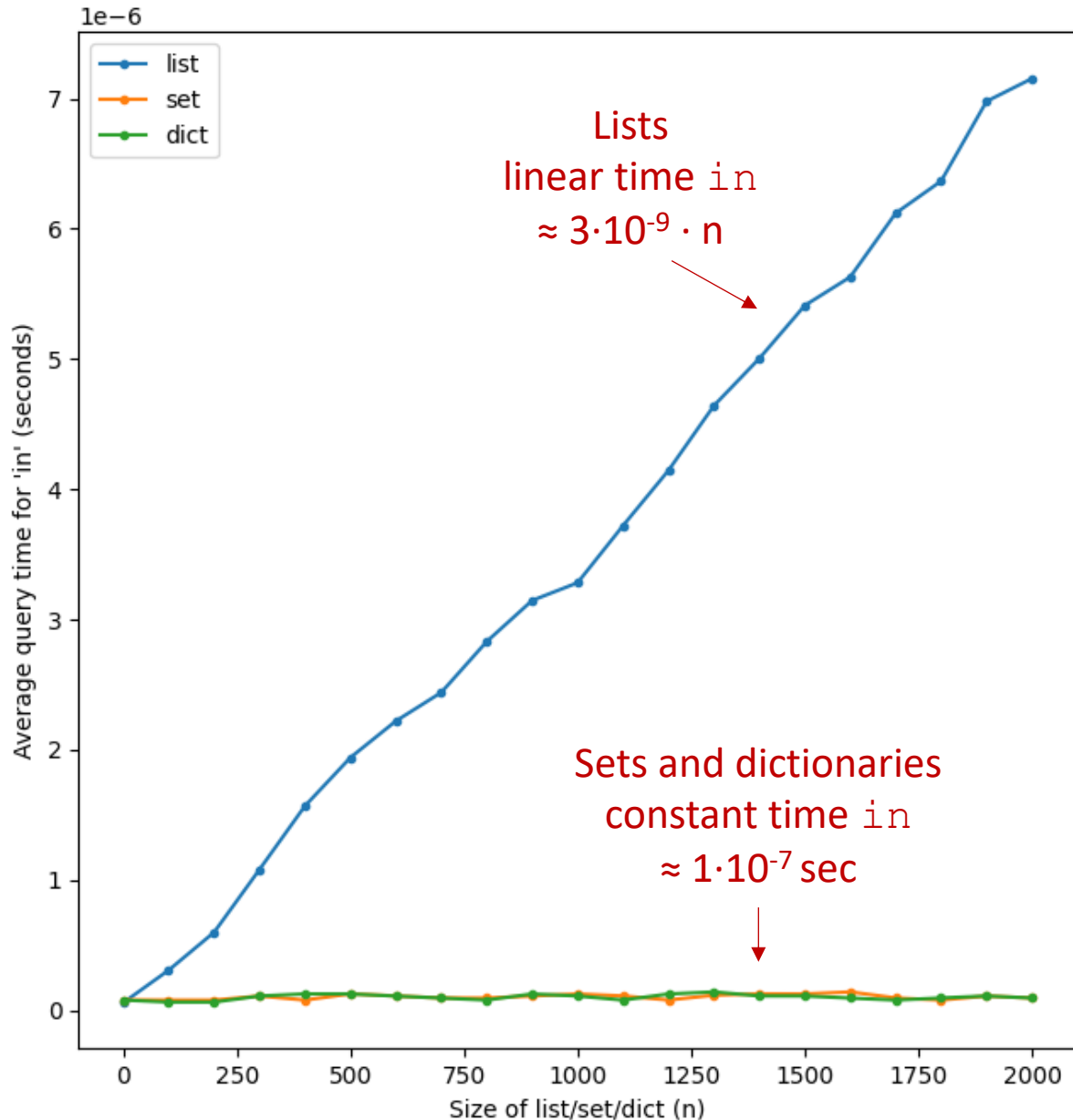
`'abc' . __hash__()` could e.g. return 624337162
`(624337162) . __hash__()` would also return 624337162

- **All built-in immutable types are hashable.**
In particular tuples of immutable values are hashable.
I.e. nested tuples like `((('a'), 'b'), ('c', ('d', 'e')))`
representing trees can be used as dictionary keys or stored in a set

Sketch of internal set implementation



Membership queries `in`



in.py

```
from random import shuffle, choices
from time import time
import matplotlib.pyplot as plt

time_L = []
time_S = []
time_D = []

ns = [1, *range(100, 2001, 100)]

for n in ns:
    print(f'{n = }')
    L = list(range(n))
    shuffle(L)
    S = set(L)
    D = {value: 42 for value in L}

    queries = choices(L, k=1_000_000)
    for X, times in [(L, time_L), (S, time_S), (D, time_D)]:
        count = 0
        start = time()
        for q in queries:
            if q in X:
                count += 1
        end = time()
        times.append((end - start) / len(queries))

plt.plot(ns, time_L, '-.', label='list')
plt.plot(ns, time_S, '-.', label='set')
plt.plot(ns, time_D, '-.', label='dict')
plt.xlabel('Size of list/set/dict (n)')
plt.ylabel('Average query time for "in" (seconds)')
plt.legend()
plt.show()
```



Avoid using `in` repeatedly on long lists

Module `collections` (container datatypes)

- Python builtin containers for data: `list`, `tuple`, `dict`, and `set`.
- The module `collections` provides further alternatives (but these are not part of the language like the builtin containers)

<code>deque</code>	double ended queue
<code>namedtuple</code>	tuples allowing access to fields by name
<code>Counter</code>	special dictionary to count occurrences of elements
<code>...</code>	

deque – double ended queues

- Extends lists with efficient updates at the front
-  Inserting at the front of a standard Python list takes linear time in the size of the list – very slow for long lists

Python shell

```
> L = list()
> L.append(1)
> L.append(2)
> L.insert(0, 0) # insert at the front
> L.insert(0, -1) # slow for long lists
> L.insert(0, -2)
> L
| [-2, -1, 0, 1, 2]
```

```
> from collections import deque
> d = deque() # create empty deque
> d.append(1)
> d.append(2)
> d.appendleft(0) # efficient
> d.appendleft(-1)
> d.appendleft(-2)
> d
| deque([-2, -1, 0, 1, 2])
> for e in d: print(e, end=', ')
| -2, -1, 0, 1, 2,
```



Since
Python 3.7
dataclass
better choice

namedtuple – tuples with field names

- Compromise between `tuple` and `dict`, can increase code readability

Python shell

```
> person = ('Donald Duck', 1934, '3 feet') # as tuple
> person[1] # not clear what is accessed
| 1934
> person = {'name': 'Donald Duck', 'appeared': 1934, 'height': '3 feet'} # as dict
> person['appeared'] # more clear what is accessed, but ['...'] overhead
| 1934
> from collections import namedtuple
> Person = namedtuple('Person', ['name', 'appeared', 'height']) # create new type
> person = Person('Donald Duck', 1934, '3 feet') # as namedtuple
> person
| Person(name='Donald Duck', appeared=1934, height='3 feet')
> person.appeared # short and clear
| 1934
> person[1] # still possible
| 1934
```

Counter – dictionaries for counting

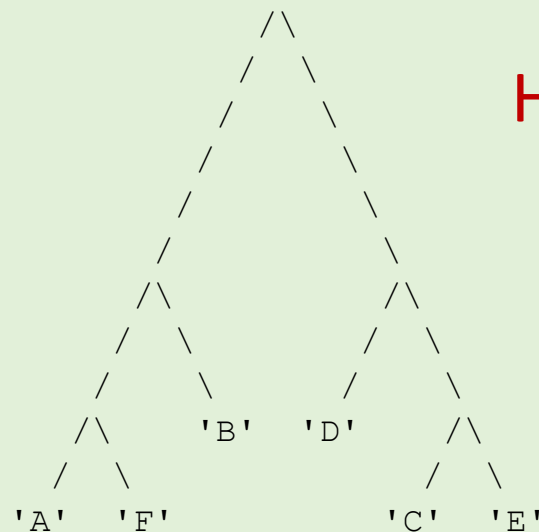
Python shell

```
> from collections import Counter
> fq = Counter('abracadabra') # create new counter from a sequence
> fq
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}) # frequencies of the letters
> fq['a']
5
> fq.most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
> fq['x'] += 5 # increase count of 'x', also valid if 'x' not in Counter yet
> Counter('aaabbbcc') - Counter('aabdd') # counters can be subtracted
Counter({'b': 2, 'c': 2, 'a': 1})
> Counter([1, 2, 1, 3, 4, 5]) + Counter([3, 3, 3]) # counters can be added
Counter({3: 4, 1: 2, 2: 1, 4: 1, 5: 1})
> T = 'AfD adsf dsa f dsaf daf dsaf DSA fda f SA dsa f dsa fdsa f dsAf sAf f dsaf'
> Counter(T.lower().split()).most_common(3)
[('f', 5), ('dsa', 4), ('dsaf', 4)]
```

Handin 3 & 4 – Triplet distance (Dobson, 1975)

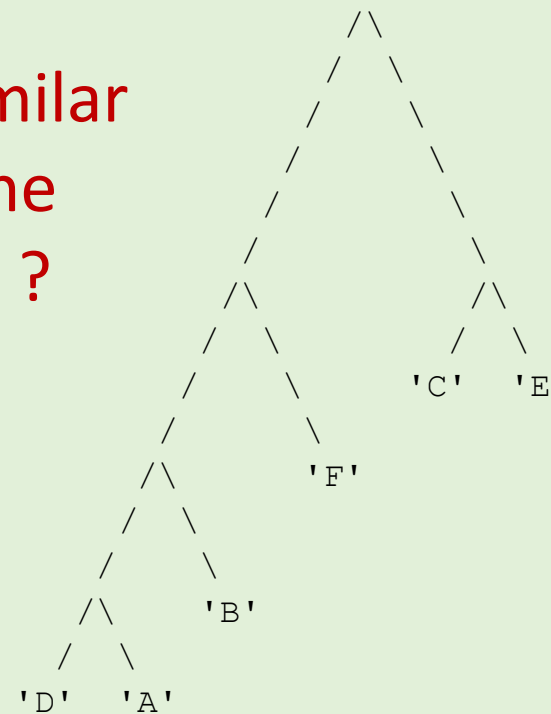
((('A', 'F'), 'B'), ('D', ('C', 'E')))) (((('D', 'A'), 'B'), 'F'), ('C', 'E')))

(a)



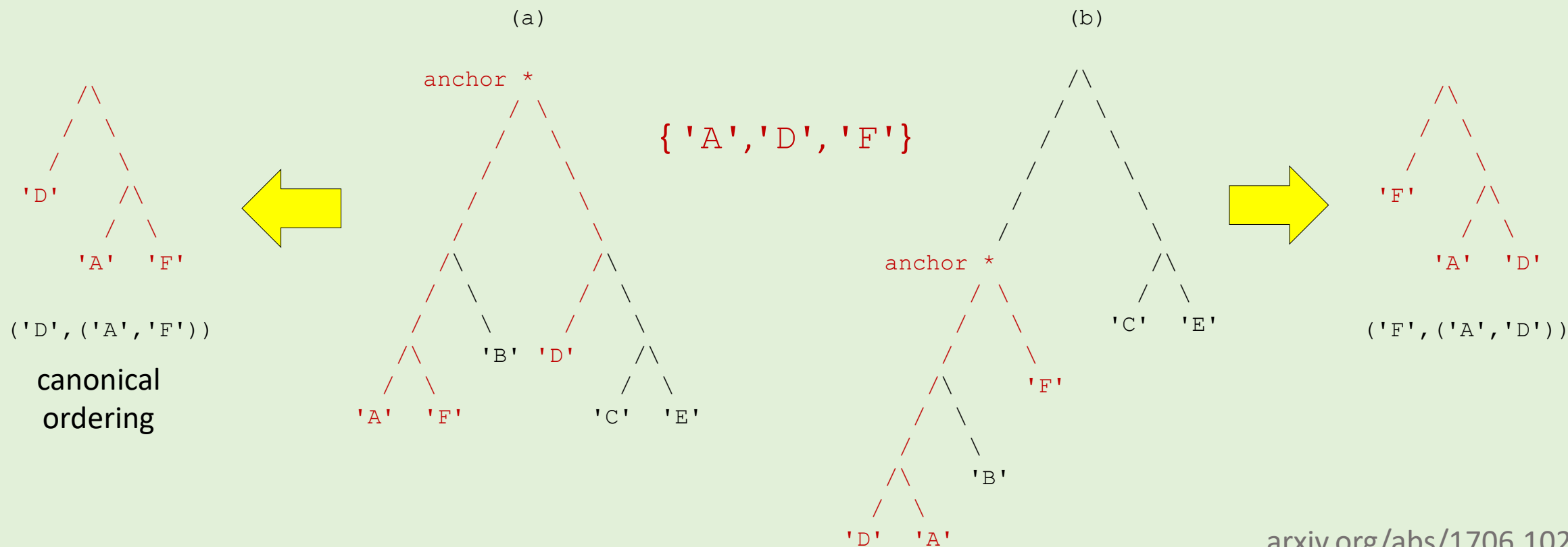
How similar
are the
trees ?

(b)



Handin 3 & 4 – Triplet distance (Dobson, 1975)

Consider all $\binom{n}{3}$ subsets of size three, and count how many do not have identical substructure (topology) in the two trees.



Functions

- functions
- return
- scoping
- arguments
- keyword arguments
- *, **, ,
- global variables

(Simple) functions

- You can define your own functions using:

def *function-name* (*var*₁, ..., *var*_k):
 body code

*var*₁, ..., *var*_k are the *formal parameters*

- If the body code executes

return expression

the result of *expression* will be returned by the function. If expression is omitted or the body code terminates without performing `return`, then `None` is returned

- When *calling* a function ***name*** (*value*₁, ..., *value*_k) body code is executed with *var*_{*i*}=*value*_{*i*}

Python shell

```
> def sum3(x, y, z):  
    return x + y + z  
  
> sum3(1, 2, 3)  
| 6  
  
> sum3(5, 7, 9)  
| 21  
  
> def powers(L, power):  
    P = [x**power for x in L]  
    return P  
  
> powers([2,3,4], 3)  
| [8, 27, 64]
```

Questions – `poly(3, "10", '3')` ?

```
def poly(z, x, y):  
    return z * x + y
```

- a) 33
- b) 1010103
- c) '33'
-  d) '1010103'
- e) TypeError
- f) Don't know

Why functions ?

Some other Python language features helping structuring programs

- Object orientation
- Modules
- Decorators
- Context managers
- Exceptions
- Doc strings
- doctest

- Avoid writting the same code multiple times, *re-usability*
- Be able to *name a functionality*
- Clearly state the functionality of a piece of code, *abstraction*:
Input = arguments, *output* = return value (and/or side effects)
- *Encapsulate* code with clear interface to the dependency to the outside world/code
- Share functionality in modules/libraries/packages with other users, *code sharing*
- Increase *readability* of code, smaller independent blocks of code
- Easier systematically *testing* of code
- ...

Local variables in functions

- The formal arguments and variables assigned to in the body of a function are created as temporary *local variables*

Global variables		Local variables	
sum3	<function>	x	4
a	3	y	5
y	42	z	6
		a	9
		b	15

state just before return b

Python shell

```
> def sum3(x, y, z):  
    a = x + y  
    b = a + z  
    return b  
  
> a = 3  
> y = 42  
> w = sum3(4, 5, 6)  
> w  
| 15  
> a  
| 3  
> b  
| NameError: name 'b' is not defined  
> x  
| NameError: name 'x' is not defined  
> y  
| 42  
> sum3  
| <function sum3 at 0x0356DA98>
```

Global variables

- Variables in function bodies that are only read, are considered access to *global variables*

Python shell

```
> prefix = 'The value is'
> def nice_print(x):
    print(prefix, x)
> nice_print(7)
| The value is 7
> prefix = 'Value ='
> nice_print(42)
| Value = 42
```

Global variables		Local variables	
nice_print	<function>	x	42
prefix	'Value ='		

state just before returning from 2nd nice_print

Global vs local variables

- If a function contains an assignment to a variable, the variable is local throughout the function – also before the first assignment

Python shell

```
> x = 42
> def f():
    print(x)    # refers to local variable
    x = 7       # x declared local variable
> f()
| UnboundLocalError: local variable 'x' referenced before assignment
```



global

- Global variables that should be updated in the function body must be declared global in the body:

`global` *variable, variable, ...*

- Note: If you only need to read a global variable, it is not required to be declared global (but would be polite to the readers of your code)

Since `counter` assigned in body, `counter` will be considered to be a local variable

Python shell

```
> counter = 1

> def counted_print(x):
    global counter
    print(f'({counter}) {x}')
    counter += 1

> counted_print(7)
| (1) 7
> counted_print(42)
| (2) 42

> def counted_print(x):
    print(f'({counter}) {x}')
    counter += 1

> counted_print(7)
| UnboundLocalError: local variable
  'counter' referenced before
  assignment
```



Question – What value is printed ?

```
x = 1
def f(a):
    global x
    x = x + 1
    return a + x
print(f(2) + f(4))
```

a) 6

b) 7

c) 8

d) 9

e) 10



f) 11

g) 12

h) Don't know

Arbitrary number of arguments

- If you would like your function to be able to take a variable number of additional arguments in addition to the required, add a **variable* as the *last* argument
- In a function call *variable* will be assigned a tuple with all the additional arguments

Python shell

```
> def my_print(x, y, *L):  
    print('x =', x)  
    print('y =', y)  
    print('L =', L)  
  
> my_print(2, 3, 4, 5, 6, 7)  
| x = 2  
| y = 3  
| L = (4, 5, 6, 7)  
  
> my_print(42)  
| TypeError: my_print() missing 1  
| required positional argument: 'y'
```

Unpacking a list of arguments in a function call

- If you have list L (or tuple) containing the arguments to a *function call*, you can unpack them in the function call using $*L$

```
L = [x, y, z]
f(*L)
```

is equivalent to calling

```
f(L[0], L[1], L[2])
```

i.e.

```
f(x, y, z)
```

- Note that $f(L)$ would pass a single argument to f , namely a list
- In a function call several $*$ expressions can appear, e.g. $f(*L1, x, *L2, *L3)$

Python shell

```
> import math
> def norm(x, y):
    return math.sqrt(x * x + y * y)
> norm(3, 5)
| 5.830951894845301
> point = (3, 4)
> print(*point, sep=':')
| 3:4
> norm(point)
| TypeError: norm() missing 1 required positional argument: 'y'
> norm(*point)
| 5.0
> def dist(x0, y0, x1, y1):
    return math.sqrt((x1 - x0) ** 2 + (y1 - y0) ** 2)
> p = 3, 7
> q = 7, 4
> dist(p, q)
| TypeError: dist() missing 2 required positional arguments: 'x1' and 'y1'
> dist(*p, *q)
| 5.0
```

Question – How many arguments should f take ?

```
a = [1, 2, 3]
b = [4, 5]
c = (6, 7, 8)
d = (9, 10)
f(*a, b, c, *d)
```

a) 4

b) 5

c) 6



d) 7

e) 8

f) 9

g) 10

h) Don't know

Question – What is `list(zip(*zip(*L)))` ?

`L = [[1, 2, 3], [4, 5], [6, 7, 8]]`

a) `[([([1, 2, 3],),), ([4, 5],), ([6, 7, 8],),)]`

b) `[((1, 4, 6),), ((2, 5, 7),)]`



c) `[(1, 2), (4, 5), (6, 7)]`

d) `[(1, 2, 3), (4, 5), (6, 7, 8)]`

e) `[([1, 2, 3], [4, 5], [6, 7, 8])]`

f) Don't know

Python shell

```
> list(zip((1, 2, 3), (4, 5, 6)))  
| [(1, 4), (2, 5), (3, 6)]
```

Keyword arguments

- Previously we have seen the following (strange) function calls

```
print(7, 14, 15, sep=':', end='')  
enumerate(my_list, start=1)
```

- name*= refers to one of the formal arguments, known as a **keyword argument**
- A *name* can appear at most once in a function call
- In function calls keyword arguments must follow positional arguments
- Improves readability when arguments have non-obvious order

```
complicated_function(  
    name='Mickey',  
    city='Duckburg',  
    state='Calisota',  
    occupation='Detective',  
    gender='Male'  
)
```

Python shell

```
> def sub(x, y):  
    return x - y  
  
> sub(9, 4)  
| 5  
  
> sub(y=9, x=4)  
| -5
```

Keyword arguments, default values

Python shell

```
> def my_print(a, b, c=5, d=7):  
    print(f'{a=} {b=} {c=} {d=}')  
  
> my_print(2, d=3, b=4)  
| a=2, b=4, c=5, d=3
```

- When calling a function arguments can be omitted if the corresponding arguments in the function definition have default values *argument=value*
- Arguments with default values must be after those without
- Default arguments are e.g. also available in Javascript, C++ and C#, and keyword arguments in e.g. Javascript and C#s

Question – What is `f(6, z=2)` ?

```
def f(x, y=3, z=7):  
    return x + y + z
```

- a) 10
-  b) 11
- c) 16
- d) `TypeError: f() missing 1 required positional argument: 'y'`
- e) Don't know

Keyword arguments, mutable default values

- Be careful: Default value will be shared among calls (which can be useful)

The Python Language Reference 8.7 Function definitions

“Default parameter values are evaluated from left to right **when the function definition is executed**. This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use None as the default”

Python shell

```
> def list_append(e, L=[]):  
    L.append(e)  
    return L  
  
> list_append('x', ['y', 'z'])  
| ['y', 'z', 'x']  
> list_append('a')  
| ['a']  
> list_append('b')  
| ['a', 'b']  
> list_append('c')  
| ['a', 'b', 'c']
```



Python shell

```
> def list_append(e, L=None):  
    if L == None:  
        L = []  
    L.append(e)  
    return L  
  
> list_append('x', ['y', 'z'])  
| ['y', 'z', 'x']  
> list_append('a')  
| ['a']  
> list_append('b')  
| ['b']  
> list_append('c')  
| ['c']
```

Function call, dictionary of keyword arguments

- If you happen to have a *dictionary* containing the keyword arguments you want to pass to function, you can give all dictionary items as arguments using the single argument ***dictionary*

Python shell

```
> print(3, 4, 5, sep=':', end='#\n')  
| 3:4:5#  
> print_kwarg = {'sep': ':', 'end': '#\n'}  
> print(3, 4, 5, **print_kwarg)  
| 3:4:5#
```

Function definition, arbitrary keyword arguments

- If you want a function to accept arbitrary keyword arguments, add an argument `**argument` to the function definition
- When the function is called *argument* will be assigned a dictionary containing the excess keyword arguments

Python shell

```
> def my_print(a, b=3, **c):  
    print('a =', a)  
    print('b =', b)  
    print('c =', c)  
  
> my_print(x=27, y=42, a=7)  
| a = 7  
| b = 3  
| c = {'x': 27, 'y': 42}
```

Example

Python shell

```
> L1 = [1, 'a']
> L2 = ['b', 2, 3]
> D1 = {'y':4, 's':10}
> D2 = {'t':11, 'z':5.0}

> def f(a, b, c, d, e, *f, q=0, x=1, y=2, z=3, **kw):
    print(f'{a=}, {b=}, {c=}, {d=}, {e=}, {f=}\n'
          f'{q=}, {x=}, {y=}, {z=}, {kw=}')
> f(7, *L1, 9, *L2, x=7, **D1, w=42, **D2)
| a=7, b=1, c=a, d=9, e=b, f=(2, 3)
| q=0, x=7, y=4, z=5.0, kw={'w': 42, 's': 10, 't': 11}
```

non-keyword arguments must appear before keyword arguments

all arguments must have distinct names

A confusing example

Python shell

```
> def f(*a, **kw):  
    print(f'{a=} {kw=}')  
  
> f(a=42)    # no positional arguments  
| a=() kw={'a': 42}
```

Forwarding function arguments

- `*` and `**` can e.g. be used to forward (unknown) arguments to other function calls

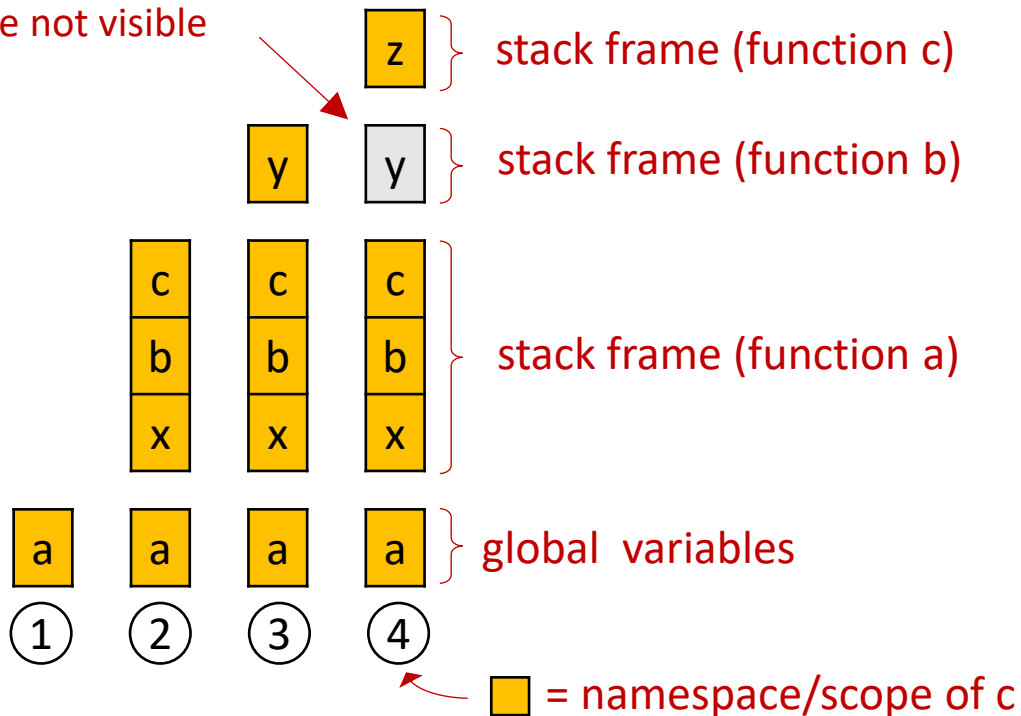
Python shell

```
> def my_print(*positional_arguments, sep=':', **keyword_arguments):  
    print(*positional_arguments, sep=sep, **keyword_arguments)  
  
> my_print(7, 42)  
| 7:42  
  
> my_print('x', 'y', end='<')  
| x:y<  
  
> my_print('x', 'y', sep='_')  
| x_y
```

Local function definitions and namespaces

- Function definitions can contain (nested) local function definitions, only accessible inside the function
- static/lexical scoping*, i.e. can see from the code which variables are in scope

in c, b's local variables
are not visible



Python shell

```
> def a(x):  
    def b(y):  
        print(f'b: {y=} {x=}')  
        c(y + 1)  
    def c(z):  
        print(f'c: {z=} {x=}')  
        print(f'a: {x=}')  
        b(x + 1)  
  
> a(42)  
a: x=42  
b: y=43 x=42  
c: z=44 x=42
```


Example – nested function definitions

Python shell

```
> def a(x):  
    def b(y):  
        print(f'Enter b ({y=}, {x=})')  
        c(y + 1)  
        print('leaving b')  
  
    def c(x): # x hides argument of function a  
        def d(z):  
            print(f'Enter d ({z=}, {x=})')  
            print('leaving d')  
  
        print(f'Enter c ({x=})')  
        d(x + 1)  
        print('leaving c')  
  
    print(f'Enter a ({x=})')  
    b(x + 1)  
    print('leaving a')
```

```
> a(5)  
| Enter a (x=5)  
| Enter b (y=6, x=5)  
| Enter c (x=7)  
| Enter d (z=8, x=7)  
| leaving d  
| leaving c  
| leaving b  
| leaving a
```

Functions must be defined before you can call them...

Python shell

```
> f()
| NameError: name 'f' is not defined
> def f():
    print('Hi')
> f()
| Hi
> def f():
    g() # g not yet defined locally
    def g():
        print('Howdy')
> f()
| UnboundLocalError: cannot access local variable 'g' where it is not associated with a value
> def f():
    def g():
        print('Howdy')
    g()
> f()
| Howdy
```



Example – nested functions and default values

Python shell

```
> def init_none(var_name):  
>     print('initializing', var_name)  
>     return None # redundant line  
  
> def f(a=init_none('a')):  
>     def g(b=init_none('b')):  
>         print('b =', b)  
>         print('a =', a)  
>         g(a + 1)  
  
| initializing a  
> f(10)  
  
| initializing b  
| a = 10  
| b = 11
```



nonlocal

- The keyword `nonlocal` makes local identifiers refer to previously bound variables in the nearest *enclosing scope excluding globals*
- `nonlocal variable, variable, ...`

```
Python shell
> x = 0

> def f():
    y = 1
    def f_helper(z):
        global x
        nonlocal y
        print(f'({x}:{y}) {z}')
        y += 1
        x += 3
    f_helper(7)
    f_helper(42)

> f()
| (0:1) 7
| (3:2) 42
> f()
| (6:1) 7
| (9:2) 42
```

Positional and keyword only arguments

- A function definition can contain `/` and `*` as arguments
- Arguments before `/` must be provided as positional arguments in a call
- Arguments after `*` cannot be positional arguments

Python shell

```
> def f(a, /, b, *, c):  
    print(a, b, c)  
  
> f(a=1, b=2, c=3)  
| TypeError: f() got some positional-only arguments passed as keyword arguments: 'a'  
> f(1, b=2, c=3)  
| 1 2 3  
> f(1, 2, c=3)  
| 1 2 3  
> f(1, 2, 3)  
| TypeError: f() takes 2 positional arguments but 3 were given
```

A note on Python and functions

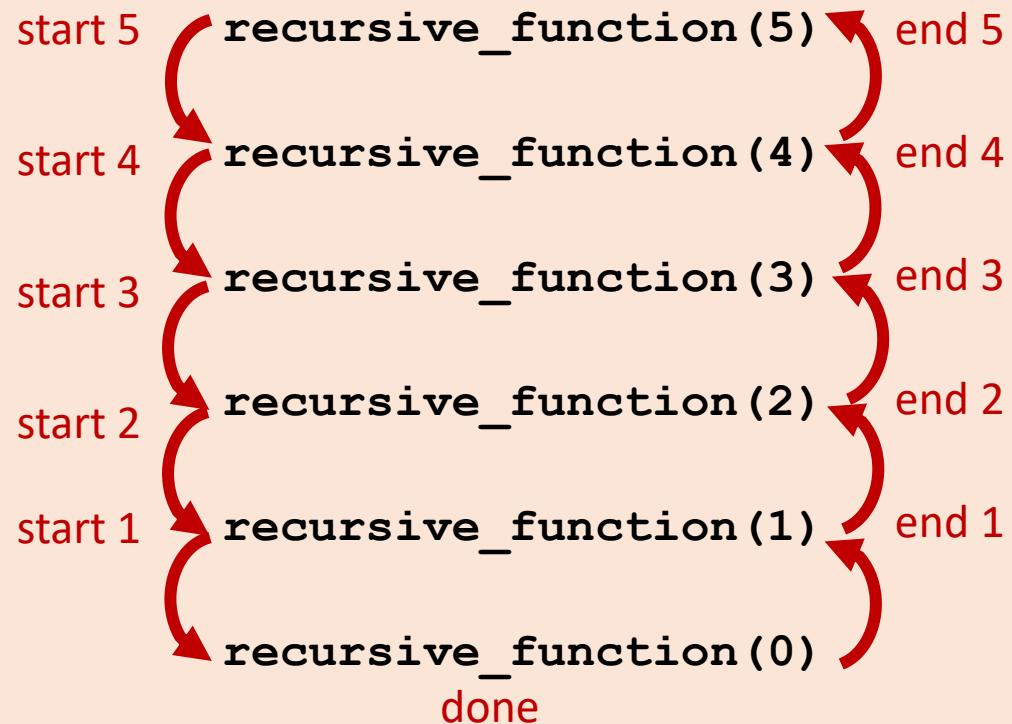
- Similarities between Python and other languages:
 - functions are widely supported (sometimes called methods and procedures)
 - scoping rules is present in many languages (but details differ)
- Python specific (but nice):
 - how to handle global, local and nonlocal variables
 - keyword arguments
 - *, **

Recursion

- symbol table
- stack frames

Recursion

Recursive function
≡
"function that calls itself"

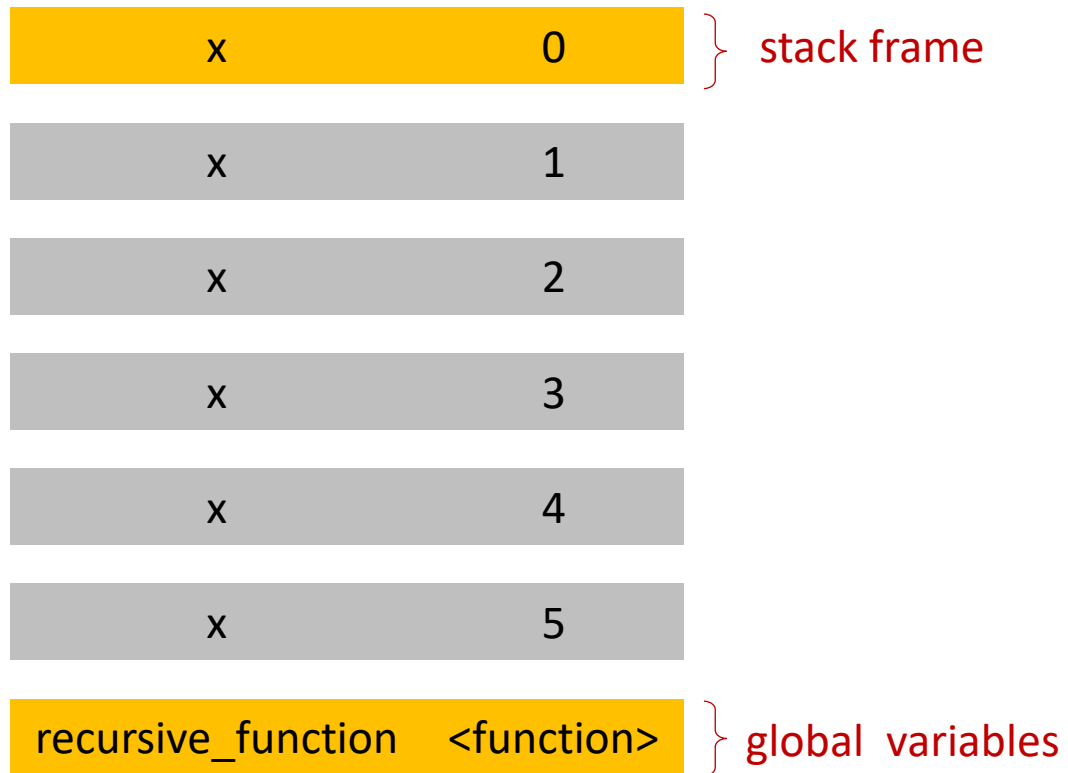


Python shell

```
> def recursive_function(x):
    if x > 0:
        print("start", x)
        recursive_function(x - 1)
        print("end", x)
    else:
        print("done")

> recursive_function(5)
| start 5
| start 4
| start 3
| start 2
| start 1
| done
| end 1
| end 2
| end 3
| end 4
| end 5
```


Recursion



Recursion stack when x = 0 is reached

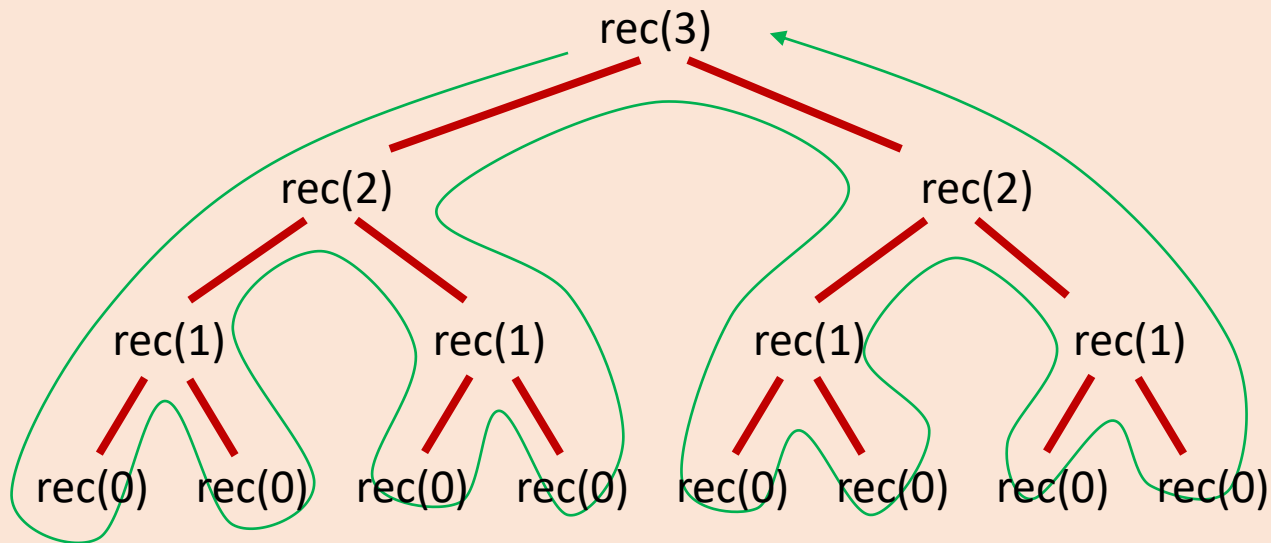
Python shell

```
> def recursive_function(x):  
    if x > 0:  
        print("start", x)  
        recursive_function(x - 1)  
    print("end", x)  
    else:  
        print("done")  
  
> recursive_function(5)  
| start 5  
| start 4  
| start 3  
| start 2  
| start 1  
| done  
| end 1  
| end 2  
| end 3  
| end 4  
| end 5
```

Python shell

```
> def rec(x):  
    if x > 0:  
        print("start", x)  
        rec(x - 1)  
        rec(x - 1)  
        print("end", x)  
    else:  
        print("done")
```

Recursion tree



Python shell

```
> rec(3)  
| start 3  
| start 2  
| start 1  
| done  
| done  
| end 1  
| start 1  
| done  
| done  
| end 1  
| end 2  
| start 2  
| start 1  
| done  
| done  
| end 1  
| done  
| end 2  
| end 3
```

Question – How many times does `rec(5)` print "done"?

Python shell

```
> def rec(x):  
    if x > 0:  
        print("start", x)  
        rec(x - 1)  
        rec(x - 1)  
        rec(x - 1)  
        print("end", x)  
    else:  
        print("done")
```

a) 3

b) 5

c) 15

d) 81

e) 125



f) $243 = 3^5$

g) Don't know

Factorial

$$n! = n \cdot \underbrace{(n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1}_{(n-1)!}$$

Observation
(recursive definition)

$$1! = 1$$

$$n! = n \cdot (n-1)!$$

factorial.py

```
def factorial(n):  
    if n <= 1:  
        return 1  
    return n * factorial(n - 1)
```

factorial.py

```
def factorial(n):  
    return n * factorial(n - 1) if n > 1 else 1
```

factorial_iterative.py

```
def factorial(n):  
    result = 1  
    for i in range(2, n + 1):  
        result *= i  
    return result
```

Binomial coefficient $\binom{n}{k}$

- $\binom{n}{k}$ = number of ways to pick k elements from a set of size n

- $$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise} \end{cases}$$

`binomial_recursive.py`

```
def binomial(n, k):  
    if k == 0 or k == n:  
        return 1  
    return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

- Unfolding computation shows $\binom{n}{k}$ 1's are added → **slow**

Readable functions ? – return early / bail out fast

- Treat simple cases first and return
- Do not put `else` after `if` ending with `return`
- Avoid unnecessary nesting of code
- 1-liners are not always the most readable code

`binomial_return_early.py`

```
def binomial(n, k): # Ugly, nested indentations and redundant else
    if k == 0:
        return 1
    else:
        if k == n:
            return 1
        else:
            return binomial(n - 1, k) + binomial(n - 1, k - 1)

def binomial(n, k): # Treat each special case first and return
    if k == 0:
        return 1
    if k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)

def binomial(n, k): # Several cases simultaneously – is test obvious?
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)

def binomial(n, k): # 1-liner, but is this the easiest to read?
    return binomial(n - 1, k) + binomial(n - 1, k - 1) if 0 < k < n else 1
```

See also *Clean Code - Handbook of Agile Software Craftsmanship*, by Robert C. Martin (2009)

Tracing the recursion

- At beginning of function call, **print** arguments
- Before returning, **print** return value
- Keep track of recursion depth in a argument to print **indentation**

binomial_trace.py

```
def binomial(n, k, indent=0):  
    print('    ' * indent + f'binomial({n}, {k})')  
    if k == 0 or k == n:  
        result = 1  
    else:  
        result = binomial(n - 1, k, indent=indent + 1) + \  
            binomial(n - 1, k - 1, indent=indent + 1)  
    print('    ' * indent + f'return {result}')  
    return result
```

Python shell

```
> binomial(4, 2)  
binomial(4, 2)  
    binomial(3, 2)  
        binomial(2, 2)  
        return 1  
        binomial(2, 1)  
            binomial(1, 1)  
            return 1  
            binomial(1, 0)  
            return 1  
        return 2  
    return 3  
    binomial(3, 1)  
        binomial(2, 1)  
            binomial(1, 1)  
            return 1  
            binomial(1, 0)  
            return 1  
        return 2  
        binomial(2, 0)  
        return 1  
    return 3  
return 6  
6
```

Binomial coefficient $\binom{n}{k}$

Observation $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$

```
binomial_factorial.py
```

```
def binomial(n, k):  
    return factorial(n) // factorial(k) // factorial(n - k)
```

- Unfolding computation shows $2n - 2$ multiplications and 2 divisions → fast
- Intermediate value $n!$ can have significantly more digits than result (bad)

Binomial coefficient $\binom{n}{k}$

Observation $\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}{k \cdot (k-1) \cdot (k-2) \cdots 1} = \binom{n-1}{k-1} \cdot \frac{n}{k}$

`binomial_recursive_product.py`

```
def binomial(n, k):  
    if k == 0:  
        return 1  
    else:  
        return binomial(n - 1, k - 1) * n // k
```

- Unfolding computation shows k multiplications and divisions $\rightarrow$ fast
- Multiplication with fractions $\geq 1 \rightarrow$ intermediate numbers limited size

Questions – Which correctly computes $\binom{n}{k}$?

Observation
$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}{k \cdot (k-1) \cdot (k-2) \cdots 1}$$

- a) binomial_A
-  b) binomial_B
- c) both
- d) none
- e) Don't know

binomial_iterative.py

```
def binomial_A(n, k):  
    result = 1  
    for i in range(k):  
        result = result * (n - i) // (k - i)  
    return result  
  
def binomial_B(n, k):  
    result = 1  
    for i in range(k) [::-1]:  
        result = result * (n - i) // (k - i)  
    return result
```

Python shell

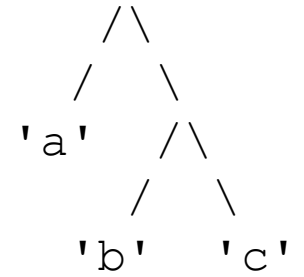
```
> binomial_A(5, 2)  
| 8  
> binomial_B(5, 2)  
| 10
```

Recursively print all leaves of a tree

- Assume a recursively nested tuple represents a tree with strings as leaves

Python shell

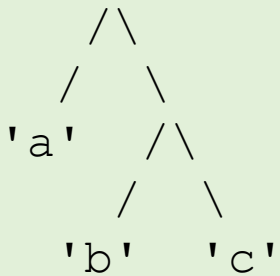
```
> def print_leaves(tree):  
    if isinstance(tree, str):  
        print('Leaf:', tree)  
    else:  
        for child in tree:  
            print_leaves(child)  
  
> print_leaves(('a', ('b', 'c')))  
| Leaf: a  
| Leaf: b  
| Leaf: c
```



Question – How many times is `print_leaves` function called in the example?

Python shell

```
> def print_leaves(tree):  
    if isinstance(tree, str):  
        print('Leaf:', tree)  
    else:  
        for child in tree:  
            print_leaves(child)  
  
> print_leaves(('a', ('b', 'c')))  
| Leaf: a  
| Leaf: b  
| Leaf: c
```



a) 3

b) 4

 c) 5

d) 6

e) Don't know

Python shell

```
> def collect_leaves_wrong(tree, leaves = set()):  
    if isinstance(tree, str):  
        leaves.add(tree)  
    else:  
        for child in tree:  
            collect_leaves_wrong(child, leaves)  
    return leaves
```



```
> def collect_leaves_right(tree, leaves = None):  
    if leaves == None:  
        leaves = set()  
    if isinstance(tree, str):  
        leaves.add(tree)  
    else:  
        for child in tree:  
            collect_leaves_right(child, leaves)  
    return leaves
```

```
> collect_leaves_wrong(('a', ('b', 'c')))  
| {'a', 'c', 'b'}  
> collect_leaves_wrong(('d', ('e', 'f')))  
| {'b', 'e', 'a', 'f', 'c', 'd'}
```

```
> collect_leaves_right(('a', ('b', 'c')))  
| {'b', 'a', 'c'}  
> collect_leaves_right(('d', ('e', 'f')))  
| {'f', 'd', 'e'}
```

Python shell

```
> def collect_leaves(tree):
    leaves = set()

    def traverse(tree):
        nonlocal leaves # can be omitted
        if isinstance(tree, str):
            leaves.add(tree)
        else:
            for child in tree:
                traverse(child)

    traverse(tree)
    return leaves

> collect_leaves(('a', ('b', 'c')))
| {'b', 'a', 'c'}
> collect_leaves(('d', ('e', 'f')))
| {'f', 'd', 'e'}
```

Maximum recursion depth ?

- Python's maximum allowed recursion depth can be increased by

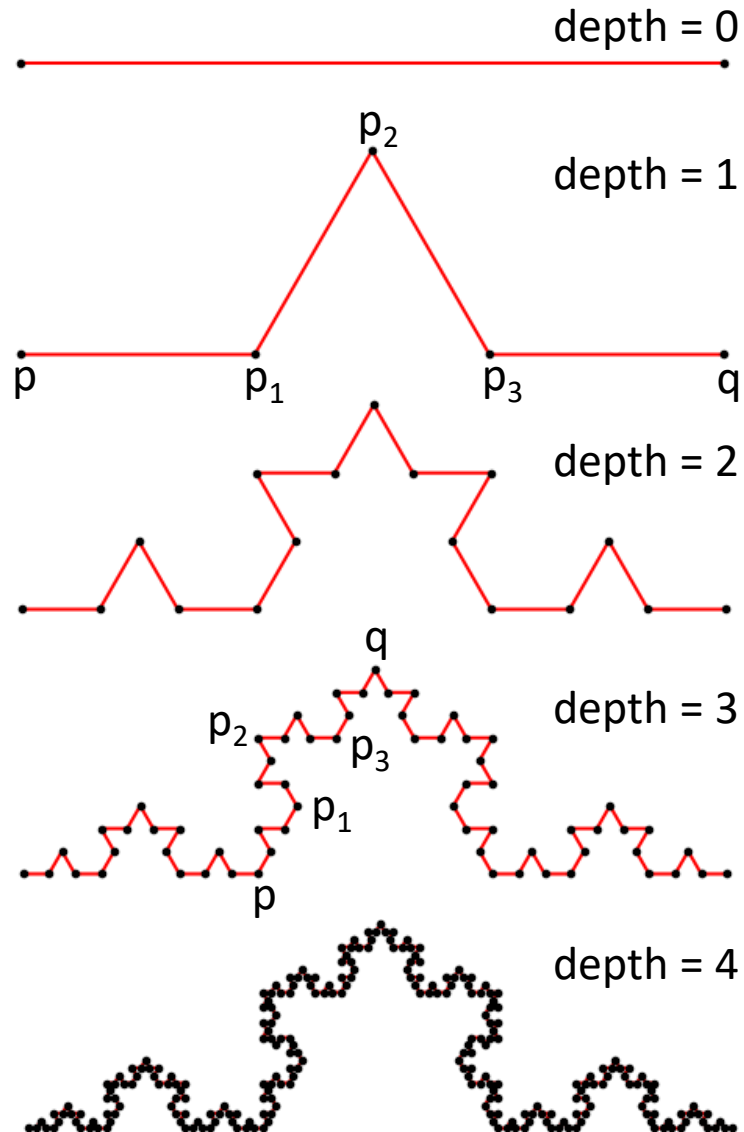
```
import sys
sys.setrecursionlimit(1500)
```

Python shell

```
> def f(x):
    print("#", x)
    f(x + 1)

> f(1)
| # 1
| # 2
| # 3
| ...
| # 975
| # 976
| # 977
| # 978
| RecursionError: maximum
  recursion depth exceeded
  while pickling an object
```

Koch Curves



`koch_curve.py`

```
import matplotlib.pyplot as plt
from math import sqrt

def koch(p, q, depth=3):
    if depth == 0:
        return [p, q]

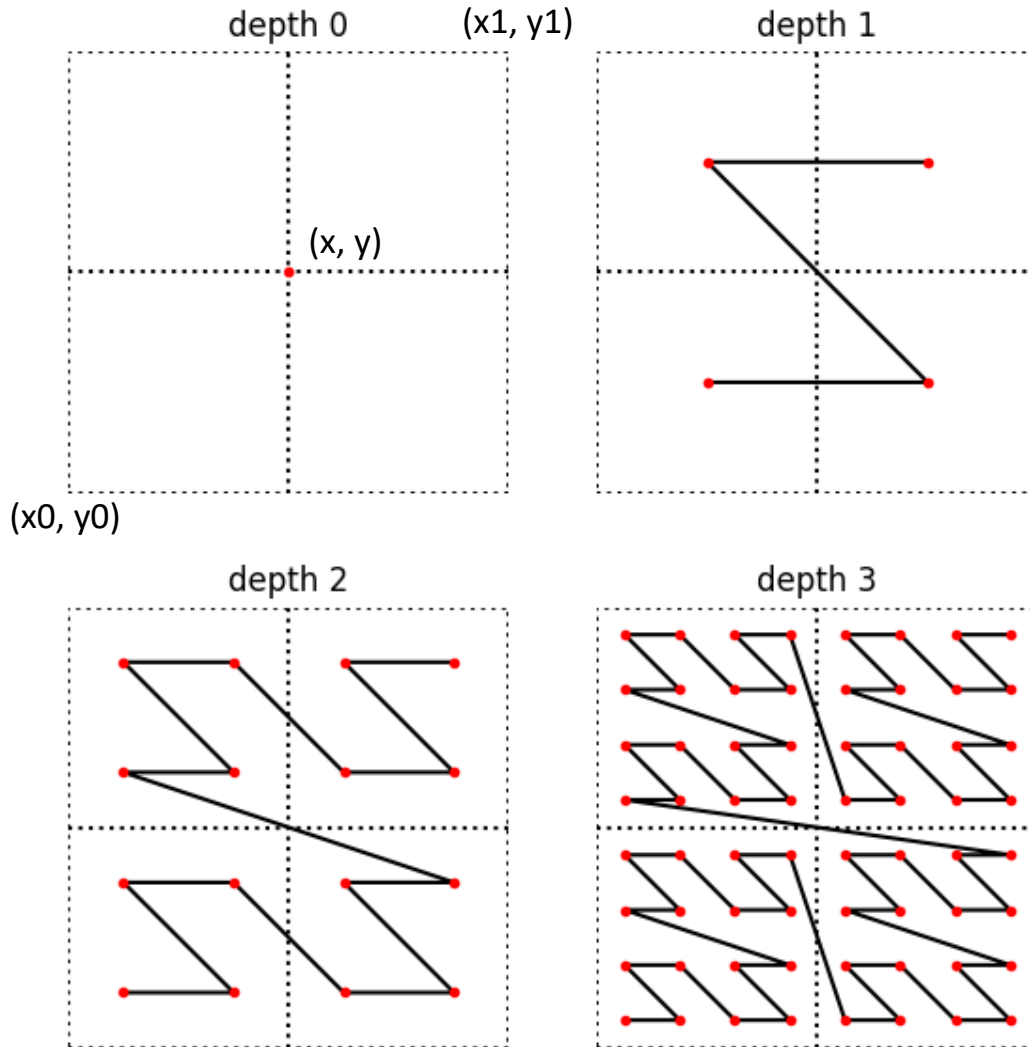
    (px, py), (qx, qy) = p, q
    dx, dy = qx - px, qy - py
    h = 1 / sqrt(12)
    p1 = px + dx / 3, py + dy / 3
    p2 = px + dx / 2 - h * dy, py + dy / 2 + h * dx
    p3 = px + dx * 2 / 3, py + dy * 2 / 3
    return (koch(p, p1, depth - 1)[: -1]
            + koch(p1, p2, depth - 1)[: -1]
            + koch(p2, p3, depth - 1)[: -1]
            + koch(p3, q, depth - 1))

points = koch((0, 0), (1, 0), depth=3)
X, Y = zip(*points)

plt.subplot(aspect='equal')
plt.plot(X, Y, 'r-')
plt.plot(X, Y, 'k.')
plt.show()
```

remove last point
(equal to first point in
next recursive call)

Z-curves



`z_curve.py`

```
import matplotlib.pyplot as plt

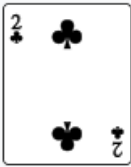
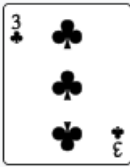
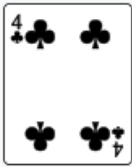
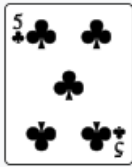
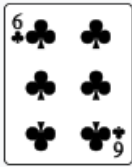
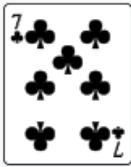
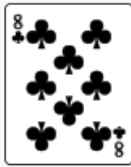
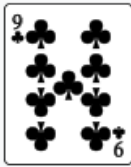
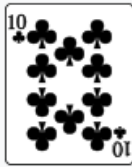
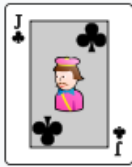
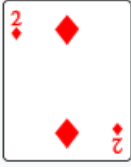
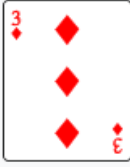
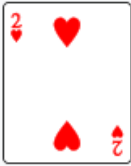
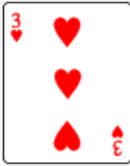
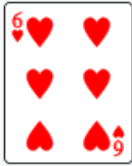
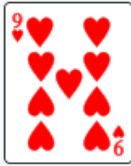
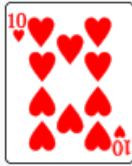
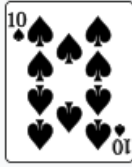
def z_curve(depth, x0=0, y0=0, x1=1, y1=1):
    x, y = (x0 + x1) / 2, (y0 + y1) / 2
    if depth == 0:
        return [(x, y)]
    return [
        *z_curve(depth - 1, x0, y0, x, y),
        *z_curve(depth - 1, x, y0, x1, y),
        *z_curve(depth - 1, x0, y, x, y1),
        *z_curve(depth - 1, x, y, x1, y1)
    ]

for depth in range(4):
    X, Y = zip(*z_curve(depth))
    plt.subplot(2, 2, 1 + depth, aspect='equal')
    plt.title(f'depth {depth}')
    plt.axis('off')
    plt.axis([0, 1, 0, 1])
    plt.plot(
        [0,1,1,0,0], [0,0,1,1,0], 'k:', # dash box
        [0.5,0.5], [0,1], 'k:', # dash vertical
        [0,1], [0.5,0.5], 'k:', # dash horizontal
        X, Y, 'k-', # Z-curve
        X, Y, 'r.', # Z-curve points
    )
plt.show()
```

Recursion and iteration

- algorithm examples

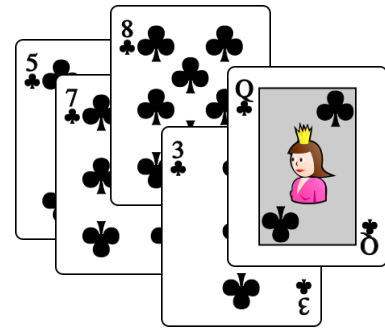
Standard 52-card deck

	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
Clubs													
Diamonds													
Hearts													
Spades													

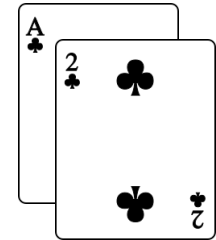
Selection sort

selection_sort.py

```
def selection_sort(L):  
    unsorted = L[:]  
    result = []  
  
    while unsorted:  
        e = min(unsorted)  
        unsorted.remove(e)  
        result.append(e)  
  
    return result
```



unsorted

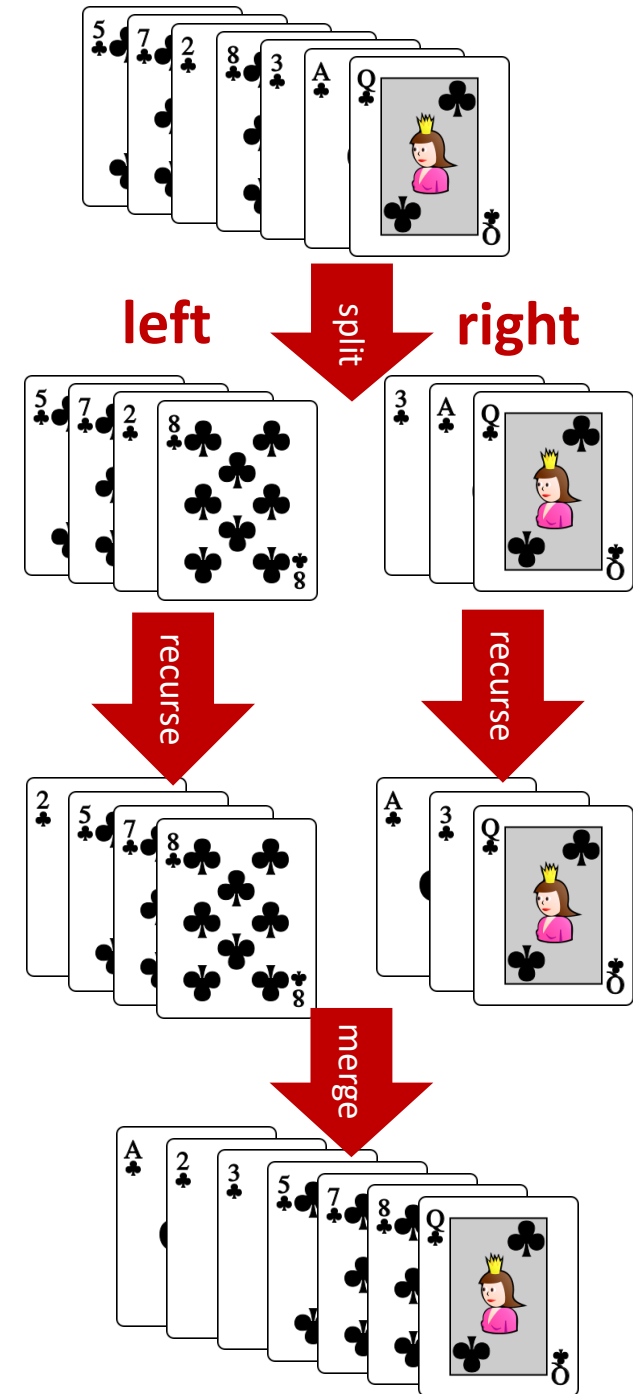


**sorted
result**

- `min` and `.remove` scan the remaining `unsorted` list for each element moved to `result`
- order $|L|^2$ comparisons

Sorting a pile of cards (Merge sort)

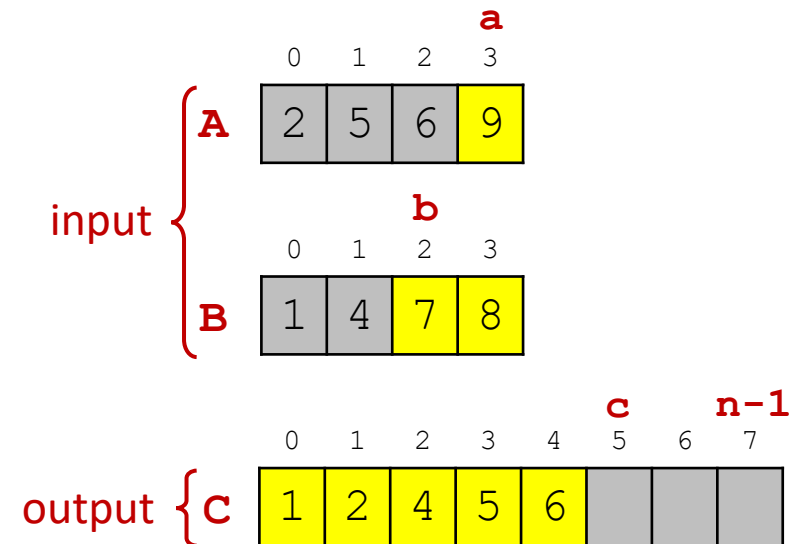
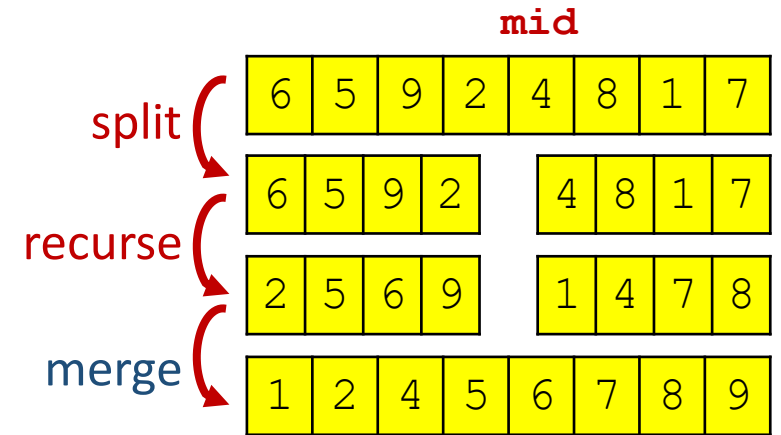
- If one card in pile, i.e. pile is sorted
- Otherwise
 - 1) Split pile into two piles, **left** and **right**, of approximately same size
 - 2) Sort **left** and **right** recursively (independently)
 - 3) Merge **left** and **right** (which are sorted)



merge_sort.py

```
def merge_sort(L):  
    n = len(L)  
    if n <= 1:  
        return L[:]  
    mid = n // 2  
    left, right = L[:mid], L[mid:]  
    return merge(merge_sort(left), merge_sort(right))
```

```
def merge(A, B):  
    n = len(A) + len(B)  
    C = n * [None]  
    a, b = 0, 0  
    for c in range(n):  
        if a < len(A) and (b == len(B) or A[a] < B[b]):  
            C[c] = A[a]  
            a = a + 1  
        else:  
            C[c] = B[b]  
            b = b + 1  
    return C
```



Question – Depth of recursion for 52 elements

a) 1

b) 2

c) 3

d) 4

e) 5

f) 6

 g) 7

h) 8

i) 9

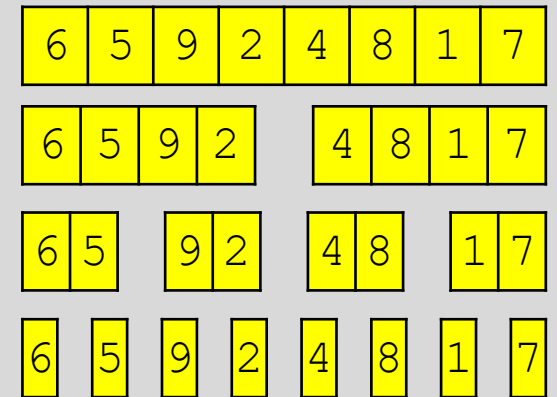
j) 10

k) Don't know

Max recursive subproblem size

$52 \rightarrow 26 \rightarrow 13 \rightarrow 7 \rightarrow 4 \rightarrow 2 \rightarrow 1$

Depth 4 for 8 elements



Question – Order of comparisons by Merge sort ?

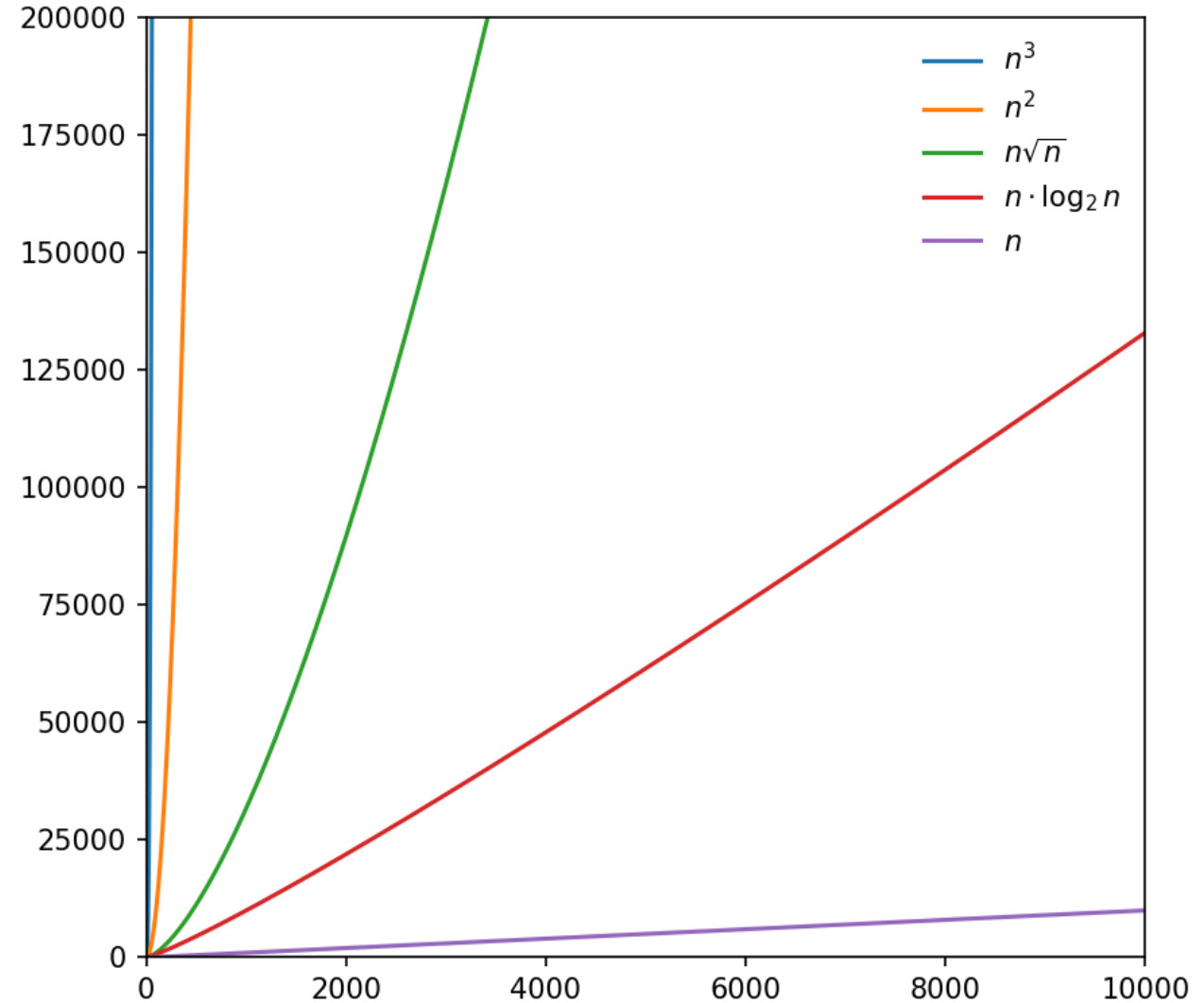
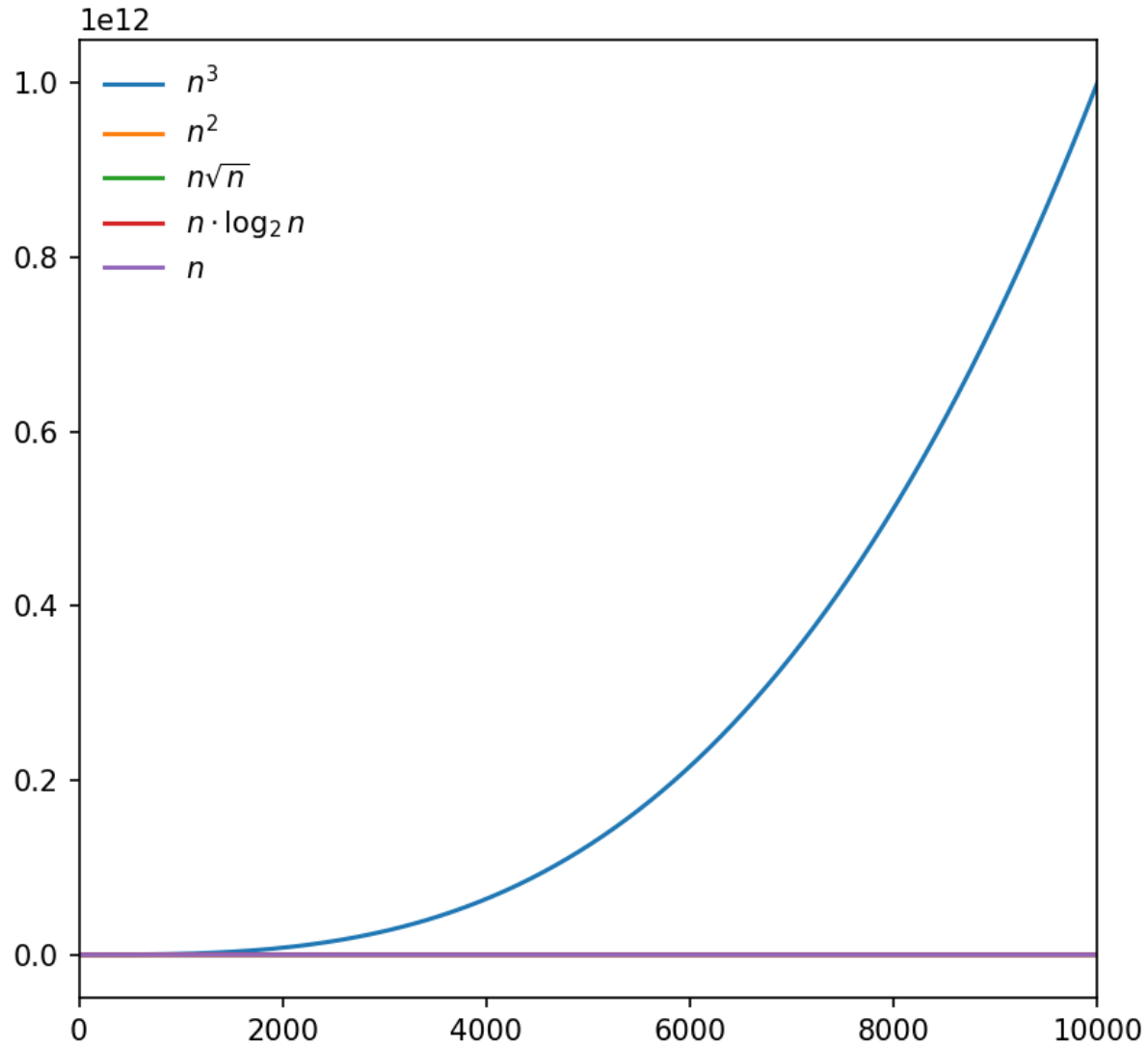
- a) $\sim n$
- b) $\sim n\sqrt{n}$
-  c) $\sim n \log_2 n$
- d) $\sim n^2$
- e) $\sim n^3$
- f) Don't know

merge_sort.py

```
def merge_sort(L):
    n = len(L)
    if n <= 1:
        return L[:]
    else:
        mid = n // 2
        left, right = L[:mid], L[mid:]
        return merge(merge_sort(left), merge_sort(right))

def merge(A, B):
    n = len(A) + len(B)
    C = n * [None]
    a, b = 0, 0
    for c in range(n):
        if a < len(A) and (b == len(B) or A[a] < B[b]):
            C[c] = A[a]
            a = a + 1
        else:
            C[c] = B[b]
            b = b + 1
    return C
```


Growth of some functions



Merge sort without recursion

- Start with piles of size one
- Repeatedly merge two smallest piles

`merge_sort.py`

```
def merge_sort_iterative(L):  
    Q = [[x] for x in L]  
    while len(Q) > 1:  
        Q.insert(0, merge(Q.pop(), Q.pop()))  
    return Q[0]
```



insert at front of
list inefficient

```
from collections import deque
```

deques are a
generalization of lists
with efficient updates
at both ends

```
def merge_sort_deque(L):  
    Q = deque([[x] for x in L])  
    while len(Q) > 1:  
        Q.appendleft(merge(Q.pop(), Q.pop()))  
    return Q[0]
```

```
merge_sort_iterative([7,1,9,3,-2,5])
```

Values of Q in while-loop

```
[[7], [1], [9], [3], [-2], [5]]  
[[-2, 5], [7], [1], [9], [3]]  
[[3, 9], [-2, 5], [7], [1]]  
[[1, 7], [3, 9], [-2, 5]]  
[[-2, 3, 5, 9], [1, 7]]  
[[-2, 1, 3, 5, 7, 9]]
```

Note: Lists in Q appear in non-increasing length order,
where longest $\leq 2 \cdot$ shortest

Question – Number of iterations of while-loop ?

```
merge_sort_iterative([7, 1, 9, 3, -2, 5])
```

- a) 1
- b) 2
- c) 3
- d) 4
-  e) 5
- f) 6
- g) 7
- h) Don't know

merge_sort.py

```
def merge_sort_iterative(L):  
    Q = [[x] for x in L]  
    while len(Q) > 1:  
        Q.insert(0, merge(Q.pop(), Q.pop()))  
    return Q[0]
```

Quicksort (randomized)

quicksort.py

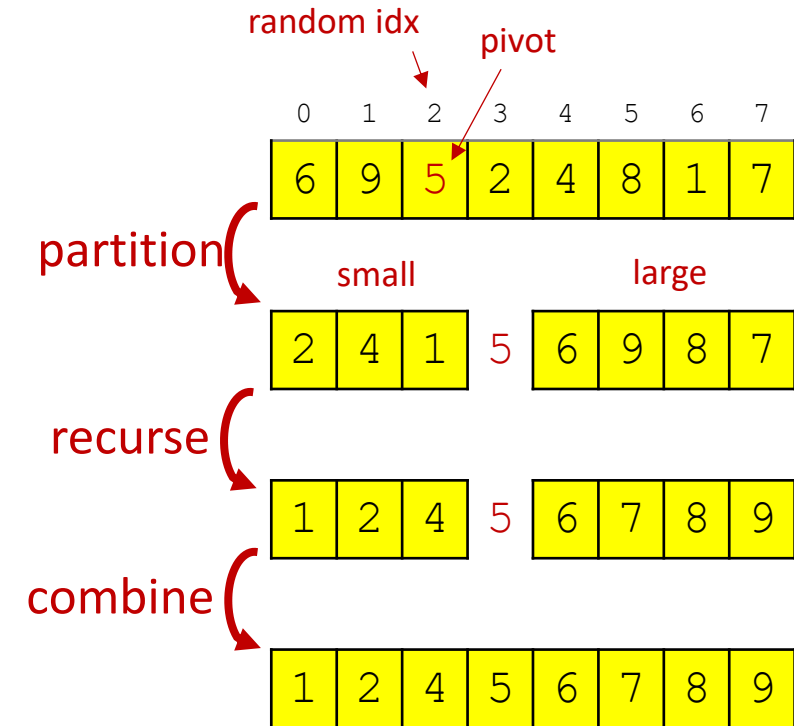
```
import random

def quicksort(L):
    if len(L) <= 1:
        return L[:]

    idx = random.randrange(len(L))
    pivot = L[idx]
    other = L[:idx] + L[idx + 1:]

    small = [e for e in other if e < pivot]
    large = [e for e in other if e >= pivot]

    return quicksort(small) + [pivot] + quicksort(large)
```



order $|L| \cdot \log_2 |L|$ comparisons, expected

Sorting comparison (single run)

tuned merge-sort (Tim-sort)
implementation in C



L	Selection sort	Merge sort Recursive	Merge sort Iterative	Merge sort Deque	Quicksort	sorted (Python builtin)
2^{10}	0.006	0.002	0.003	0.002	0.002	0.00004
2^{11}	0.02	0.004	0.006	0.000	0.003	0.0001
2^{12}	0.09	0.008	0.01	0.008	0.007	0.0003
2^{13}	0.37	0.02	0.04	0.03	0.02	0.0007
2^{14}	1.50	0.04	0.10	0.06	0.03	0.002
2^{15}	6.19	0.08	0.26	0.13	0.07	0.003
2^{16}	25.67	0.18	0.81	0.26	0.14	0.008
2^{17}	104.20	0.38	2.96	0.61	0.29	0.02
2^{18}		0.81	10.78	1.29	0.62	0.04
2^{19}		1.69	41.71	2.58	1.48	0.09
2^{20}		3.65	167.31	5.15	3.30	0.20
2^{21}		7.85		9.68	7.53	0.45
2^{22}		16.69		19.09	17.6	1.00

x 4

x 4

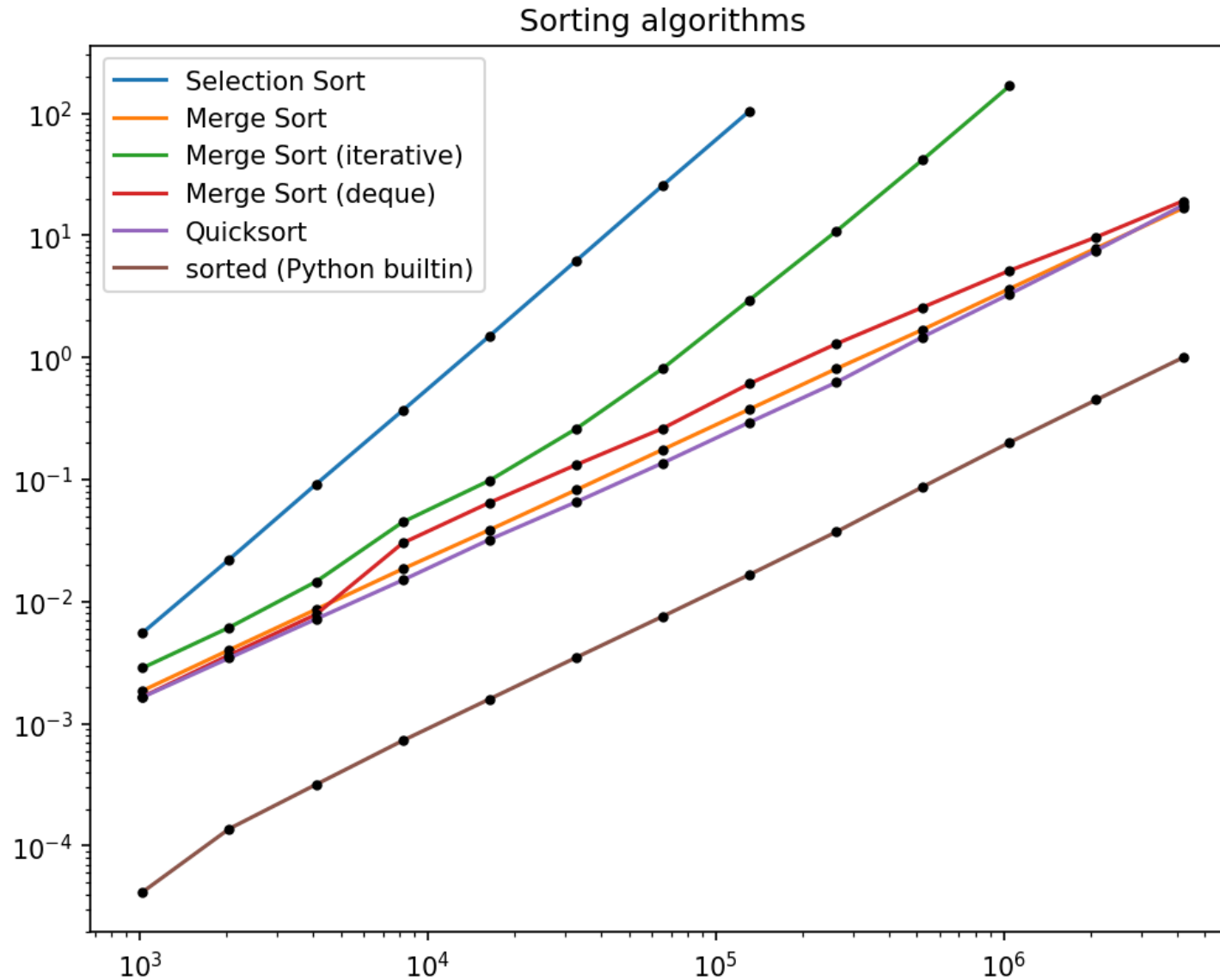
x 2

x 2

x 2

x 2

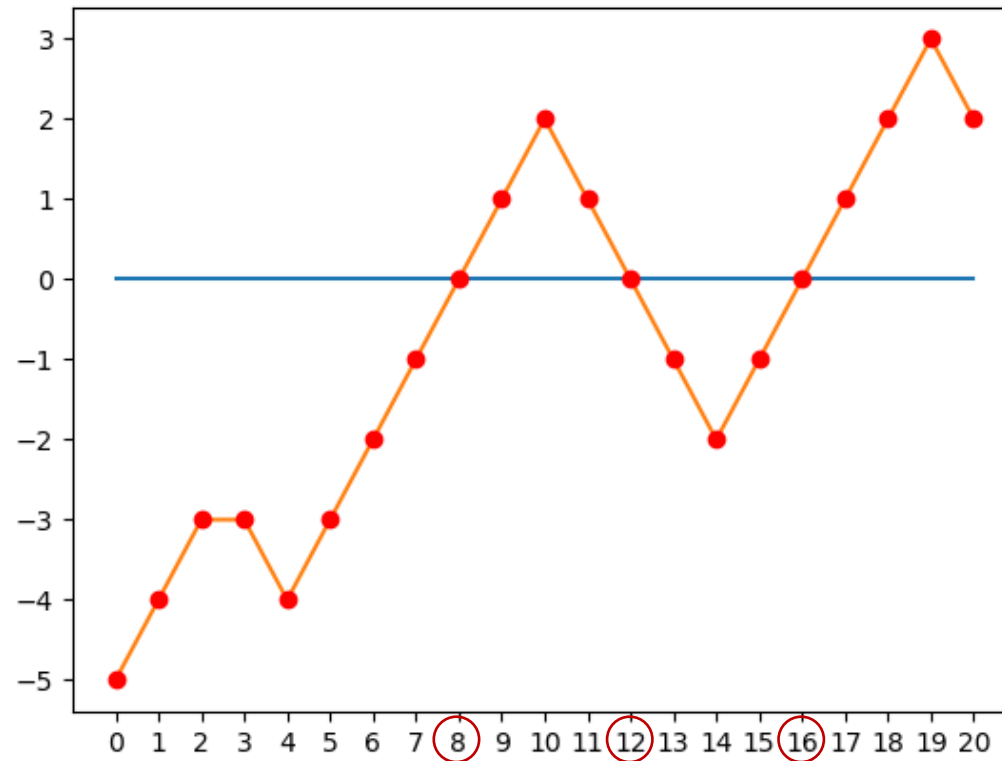
Sorting comparison



Find zero

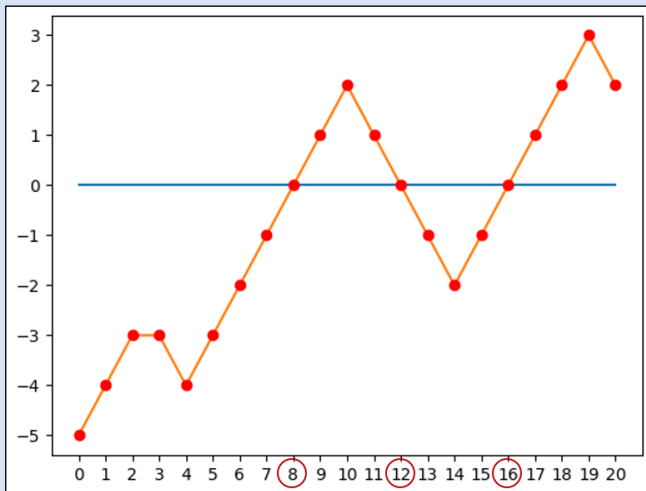
- Given a list L of integers starting with a negative and ending with a positive integer, and where $|L[i+1] - L[i]| \leq 1$, find the position of a zero in L.

$L = [-5, -4, -3, -3, -4, -3, -2, -1, 0, 1, 2, 1, 0, -1, -2, -1, 0, 1, 2, 3, 2]$



find_zero.py

```
def find_zero_loop(L):  
    i = 0  
    while L[i] != 0:  
        i += 1  
    return i  
  
def find_zero_enumerate(L):  
    for i, e in enumerate(L):  
        if e == 0:  
            return i  
  
def find_zero_index(L):  
    return L.index(0)
```



```
def find_zero_binary_search(L):  
    low = 0  
    high = len(L) - 1  
    while True: # L[low] < 0 < L[high]  
        mid = (low + high) // 2  
        if L[mid] == 0:  
            return mid  
        elif L[mid] < 0:  
            low = mid  
        else:  
            high = mid  
  
def find_zero_recursive(L):  
    def search(low, high):  
        mid = (low + high) // 2  
        if L[mid] == 0:  
            return mid  
        elif L[mid] < 0:  
            return search(mid, high)  
        else:  
            return search(low, mid)  
  
    return search(0, len(L) - 1)
```


find_zero.py

```
def find_zero_loop(L):
    i = 0
    while L[i] != 0:
        i += 1
    return i

def find_zero_enumerate(L):
    for i, e in enumerate(L):
        if e == 0:
            return i

def find_zero_index(L):
    return L.index(0)
```

Function ($ L = 10^6$)	Time, sec
find_zero_loop	0.13
find_zero_enumerate	0.10
find_zero_index	0.015
find_zero_binary_search	0.000015
find_zero_recursive	0.000088

```
def find_zero_binary_search(L):
    low = 0
    high = len(L) - 1
    while True: # L[low] < 0 < L[high]
        mid = (low + high) // 2
        if L[mid] == 0:
            return mid
        elif L[mid] < 0:
            low = mid
        else:
            high = mid

def find_zero_recursive(L):
    def search(low, high):
        mid = (low + high) // 2
        if L[mid] == 0:
            return mid
        elif L[mid] < 0:
            return search(mid, high)
        else:
            return search(low, mid)

    return search(0, len(L) - 1)
```

Greatest Common Divisor (GCD)

Notation $x \uparrow y$ denotes y is divisible by x , e.g. $3 \uparrow 12$
i.e. $y = a \cdot x$ for some integer a

Definition $\text{gcd}(m, n) = \max \{ x \mid x \uparrow m \text{ and } x \uparrow n \}$

Fact if $x \uparrow y$ and $x \uparrow z$ then $x \uparrow (y + z)$ and $x \uparrow (y - z)$

Observation
(recursive definition)

$$\text{gcd}(m, n) = \begin{cases} m & \text{if } m = n \\ \text{gcd}(m, n - m) & \text{if } m < n \\ \text{gcd}(m - n, n) & \text{if } m > n \end{cases}$$

$\text{gcd}(90, 24)$

m	n
90	24
66	24
42	24
18	24
18	6
12	6
6	6

Greatest Common Divisor (GCD)

gcd_slow.py

```
def gcd(m, n):  
    while m != n:  
        if n > m:  
            n = n - m  
        else:  
            m = m - n  
    return m
```

gcd_slow_recursive.py

```
def gcd(m, n):  
    if m == n:  
        return m  
    elif m > n:  
        return gcd(m - n, n)  
    else:  
        return gcd(m, n - m)
```

gcd.py

```
def gcd(m, n):  
    while n != 0:  
        m, n = n, m % n  
    return m
```

gcd_recursive.py

```
def gcd(m, n):  
    if n == 0:  
        return m  
    else:  
        return gcd(n, m % n)
```

gcd_recursive_one_line.py

```
def gcd(m, n):  
    return m if n == 0 else gcd(n, m % n)
```

Permutations

- Generate a list L of all permutations of a tuple

Python shell

```
> permutations(('a', 'b', 'c'))  
| [('a', 'b', 'c'), ('b', 'a', 'c'), ('b', 'c', 'a'),  
  ('a', 'c', 'b'), ('c', 'a', 'b'), ('c', 'b', 'a')]
```

permutations.py

```
def permutations(L):  
    if len(L) == 0:  
        return [L[:]] # empty tuple (ensures same type as L)  
    else:  
        P = permutations(L[1:])  
        return [p[:i] + L[:1] + p[i:] for p in P for i in range(len(L))]
```

- An implementation of `permutations` exists in the `itertools` module

Maze solver

Input

- First line #rows and #columns
- Following #rows lines contain strings containing #column characters
- There are exactly one 'A' and one 'B'
- '.' are free cells and '#' are blocked cells

Output

- Print whether there is a path from 'A' to 'B' or not

maze input

11 19

```
#####A#####  
#.....#.....#...#  
#.###.###...#.#.#.  
#...#.....#.#...#.  
#.#.####.#.#.###.#  
#.#.....#...#...#  
#.#####.###.###.  
#.#.#.....#...#...#  
#.#.#####.###.###  
#.....#...#...#  
#####B####
```

Maze solver (recursive)

maze_solver.py

```
def explore(i, j):
    global solution, visited

    if (0 <= i < n and 0 <= j < m and
        maze[i][j] != '#' and not visited[i][j]):

        visited[i][j] = True

        if maze[i][j] == 'B':
            solution = True

        explore(i - 1, j)
        explore(i + 1, j)
        explore(i, j - 1)
        explore(i, j + 1)
```

maze input

```
11 19
#####A#####
#.....#.....#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#...#...#...#...#
#####B###
```

```
def find(symbol):
    for i, row in enumerate(maze):
        j = row.find(symbol)
        if j >= 0:
            return (i, j)

n, m = [int(x) for x in input().split()]
maze = [input() for i in range(n)]

solution = False
visited = [m * [False] for i in range(n)]

explore(*find('A'))

if solution:
    print('path from A to B exists')
else:
    print('no path')
```

Maze solver (iterative)

maze_solver_iterative.py

```
def explore(i, j):
    global solution, visited

    Q = [(i, j)] # cells to visit

    while Q:
        i, j = Q.pop()
        if (0 <= i < n and 0 <= j < m and
            maze[i][j] != '#' and not visited[i][j]):

            visited[i][j] = True

            if maze[i][j] == 'B':
                solution = True

            Q.append((i - 1, j))
            Q.append((i + 1, j))
            Q.append((i, j - 1))
            Q.append((i, j + 1))
```

```
def find(symbol):
    for i, row in enumerate(maze):
        j = row.find(symbol)
        if j >= 0:
            return (i, j)

n, m = [int(x) for x in input().split()]
maze = [input() for i in range(n)]

solution = False
visited = [m*[False] for i in range(n)]

explore(*find('A'))

if solution:
    print("path from A to B exists")
else:
    print("no path")
```

Question – How difficult is the triplet project on a scale 1 – 10 ?

- a) 1 (I'm offended by how trivial the project was)
- b) 2 (very easy)
- c) 3 (a quite standard review exercise)
- d) 4 (not too complicated, got some known concepts repeated)
- e) 5 (good exercise to repeat standard programming techniques)
- f) 6 (had to use more advanced techniques in a familiar way)
- g) 7 (quite complicated, but manageable)
- h) 8 (very abstract exercise, using complicated language constructs)
- i) 9 (very complicated – barely manageable spending all my time)
- j) 10 (this is a research project – could be an MSc thesis/PhD project)
- k) 25 (this is wayyy too complicated for a university course)

Functions as objects

- lambda
- higher-order functions
- map, filter, reduce

Aliasing functions – both user defined and builtin

Python shell

```
> def square(x):  
    return x * x  
  
> square  
| <function square at 0x0329A390>  
  
> square(8)  
| 64  
  
> kvadrat = square  
| kvadrat(5)  
| 25  
  
> kvadrat  
| <function square at 0x0329A390>  
  
> len  
| <built-in function len>  
  
> length = len  
> length([1, 2, 3])  
| 3
```

Functions as values

square_or_double.py

```
def square(x):  
    return x * x  
  
def double(x):  
    return 2 * x  
  
while True:  
    answer = input('square or double ? ')  
    if answer == 'square':  
        f = square  
        break  
    if answer == 'double':  
        f = double  
        break  
  
answer = input('numbers: ')  
L_in = [int(x) for x in answer.split()]  
L_out = [f(x) for x in L_in]  
print(L_out)
```

f will refer to one of the functions
square and double refer to
call the function f is referring to
with argument x

Python shell

```
| square or double ? square  
| numbers: 3 6 7 9  
| [9, 36, 49, 81]  
  
| square or double ? double  
| numbers: 2 3 4 7 9  
| [4, 6, 8, 14, 18]
```

Functions as values and namespaces

say.py

```
def what_says(name):  
    def say(message):  
        print(name, 'says:', message)  
    return say  
  
alice = what_says('Alice')  
peter = what_says('Peter')  
  
alice('Where is Peter?')  
peter('I am here')
```

Python shell

```
| Alice says: Where is Peter?  
| Peter says: I am here
```

- `what_says` is a function returning a function (`say`)
- Each call to `what_says` with a single string as its argument **creates a new `say` function** with the current `name` argument in its namespace
- In each call to an instance of a `say` function, `name` refers to the string in the namespace when the function was created, and `message` is the string given as an argument in the call

Question – What list is printed ?

```
def f(x):  
    def g(y):  
        nonlocal x  
        x = x + 1  
        return x + y  
    return g
```

a = f(3)

b = f(6)

print([**a**(3), **b**(2), **a**(4)])

a) [7, 7, 10]

b) [7, 9, 8]



c) [7, 9, 9]

d) [7, 9, 12]

e) [7, 10, 10]

f) Don't know

map

- `map(function, list)` applies the function to each element of the sequence `list`
- `map(function, list1, ..., listk)` requires `function` to take `k` arguments, and creates a sequence with the `i`'th element being `function(list1[i], ..., listk[i])`

Python shell

```
> def square(x):  
    return x * x  
  
> list(map(square, [1,2,3,4,5]))  
| [1, 4, 9, 16, 25]  
  
> def triple_sum(x, y, z):  
    return x + y + z  
  
> list(map(triple_sum, [1,2,3], [4,5,6], [7,8,9]))  
| [12, 15, 18]  
  
> list(map(triple_sum, *zip(*(1,4,7), (2,5,8), (3,6,9))))  
| [12, 15, 18]
```

sorted

- A list `L` can be sorted using `sorted(L)`
- A user defined order on the elements can be defined by providing a function using the keyword argument `key`, that maps elements to values with some default ordering

Python shell

```
> def length_square(p):  
    x, y = p  
    return x**2 + y**2 # no sqrt  
  
> L = [(5, 3), (2, 5), (1, 9), (2, 2), (3, 4)]  
> list(map(length_square, L))  
| [34, 29, 82, 8, 25]  
> sorted(L) # default lexicographical order  
| [(1, 9), (2, 2), (2, 5), (3, 4), (5, 3)]  
> sorted(L, key=length_square) # order by length  
| [(2, 2), (3, 4), (2, 5), (5, 3), (1, 9)]
```

Question – What list does sorted produce ?

```
sorted([2, 3, -1, 5, -4, 0, 8, -6], key=abs)
```

key **2** **3** **1** **5** **4** **0** **8** **6**

- a) [-6, -4, -1, 0, 2, 3, 5, 8]
- b) [0, 2, 3, 5, 8, -1, -4, -6]
-  c) [0, -1, 2, 3, -4, 5, -6, 8]
- d) [8, 5, 3, 2, 0, -1, -4, -6]
- e) [0, 1, 2, 3, 4, 5, 6, 8]
- f) Don't know

Python shell

```
> abs(7)
| 7
> abs(-42)
| 42
```


filter

- `filter(function, list)` returns the subsequence of `list` where `function` evaluates to true
- Essentially the same as
`[x for x in list if function(x)]`

Python shell

```
> def odd(x):  
    return x % 2 == 1  
  
> filter(odd, range(10))  
| <filter object at 0x03970FD0>  
> list(filter(odd, range(10)))  
| [1, 3, 5, 7, 9]
```

reduce (in module functools)

- Python's "reduce" function is in other languages often denoted "fold"

$$\text{reduce}(f, [x_1, x_2, x_3, \dots, x_k]) = f(\dots f(f(x_1, x_2), x_3) \dots, x_k)$$

Python shell

```
> from functools import reduce  
  
> def power(x, y):  
    return x ** y  
  
> reduce(power, [2, 2, 2, 2, 2])  
| 65536
```

lambda (anonymous functions)

- If you need to define a *short* function, that *returns a value*, and the function is only *used once* in your program, then a lambda function might be appropriate:

`lambda arguments: expression`

- Creates a function with no name that takes zero or more arguments, and returns the value of the single expression

Python shell

```
> f = lambda x, y: x + y # discouraged by PEP 8, use def
> f(2, 3)
| 5
> list(filter(lambda x: x % 2, range(10)))
| [1, 3, 5, 7, 9]
```

History of lambda in programming languages

- Lambda calculus invented by Alonzo Church in 1930s
- At the heart of functional programming languages
 - LISP (1958)
 - ML (1973) and its derivatives, e.g. OCaml (1996) and F* (2011)
 - Haskell (1990), based on Miranda (1985)
- C++ got lambdas with C++11 in 2011 (26 years after 1st release)
- Java first got lambdas with Java 8 in 2014 (19 years after 1st release)
- Python has had lambdas since its inception in 1994

Examples: sorted using lambda

Python shell

```
> L = [ 'AHA', 'Oasis', 'ABBA', 'Beatles', 'AC/DC', 'B. B. King', 'Bangles', 'Alan Parsons']

> # Sort by length, secondary after input position (default, known as stable)
> sorted(L, key=len)
| ['AHA', 'ABBA', 'Oasis', 'AC/DC', 'Beatles', 'Bangles', 'B. B. King', 'Alan Parsons']

> # Sort by length, secondary alphabetically
> sorted(L, key=lambda s: (len(s), s))
| ['AHA', 'ABBA', 'AC/DC', 'Oasis', 'Bangles', 'Beatles', 'B. B. King', 'Alan Parsons']

> # Sort by most 'a's, if equal by number of 'b's, etc.
> sorted(L, key=lambda s: sorted([a.lower() for a in s if a.isalpha()] + ['~']))
| ['Alan Parsons', 'ABBA', 'AHA', 'Beatles', 'Bangles', 'AC/DC', 'Oasis', 'B. B. King']

> sorted([a.lower() for a in 'AC/DC' if a.isalpha()] + ['~'])
| ['a', 'c', 'c', 'd', '~'] # '~' is larger than all letters 'a'-'z'
```

min and max

- Similarly to `sorted`, the functions `min` and `max` take a keyword argument `key`, to map elements to values with some default ordering

Python shell

```
> max(['w', 'xyz', 'abcd', 'uv'])
| 'xyz'
> max(['w', 'xyz', 'abcd', 'uv'], key=len)
| 'abcd'
> sorted([210, 13, 1010, 30, 27, 103], key=lambda x: str(x)[::-1])
| [1010, 210, 30, 103, 13, 27]
> min([210, 13, 1010, 30, 27, 103], key=lambda x: str(x)[::-1])
| 1010
```

defaultdict (from module collections)

- An extension of the built-in dict that automatically initializes undefined items on access by calling a function (factory) to produce a default value to be inserted

`defaultdict` (*function to create default value*, *normal dict arguments*)

Python shell

```
> scores = {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2]}
> scores['Gladstone'] # access to undefined key in a standard dictionary
| KeyError: 'Gladstone'
> from collections import defaultdict
> scores = defaultdict(lambda : [], {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2]})
> scores
| defaultdict(<function <lambda> at 0x0000026460F0BBE0>, {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2]})
> scores['Gladstone'] # calls lambda without arguments to initialize scores['Gladstone']
| []
> scores
| defaultdict(<function <lambda> at ...>, {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2], 'Gladstone': []})
> scores = defaultdict(list, Mickey=[2, 3, 1], Goofy=[1, 0, 2])
> scores['daisy'].append(7) # calls list() to initialize scores['daisy']
> scores
| defaultdict(<class 'list'>, {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2], 'daisy': [7]})
```

polynomial.py

```
def linear_function(a, b):  
    return lambda x: a * x + b  
  
def degree_two_polynomial(a, b, c):  
    def evaluate(x):  
        return a * x**2 + b * x + c  
    return evaluate  
  
def polynomial(coefficients):  
    return lambda x: sum([c * x**p for p, c in enumerate(coefficients)])  
  
def combine(f, g):  
    def evaluate(*args, **kwargs):  
        return f(g(*args, **kwargs))  
    return evaluate  
  
f = linear_function(2, 3)  
for x in [0, 1, 2]:  
    print(f'f({x}) = {f(x)}')  
  
p = degree_two_polynomial(1, 2, 3)  
for x in [0, 1, 2]:  
    print(f'p({x}) = {p(x)}')  
  
print(f'{polynomial([3, 2, 1])(2) = }')  
  
h = combine(abs, lambda x, y: x - y)  
print(f'{h(3, 5) = }')
```

Python shell

```
| f(0) = 3  
| f(1) = 5  
| f(2) = 7  
| p(0) = 3  
| p(1) = 6  
| p(2) = 11  
| polynomial([3, 2, 1])(2) = 11  
| h(3, 5) = 2
```


Question – What value is $h(1)$?

`linear_combine.py`

```
def combine(f, g):  
    def evaluate(*args, **kwargs):  
        return f(g(*args, **kwargs))  
  
    return evaluate  
  
def linear_function(a, b):  
    return lambda x: a * x + b  
  
f = linear_function(2, 3)  
g = linear_function(4, 5)  
  
h = combine(f, g)  
  
print(h(1))
```

a) 5

b) 9

c) 16

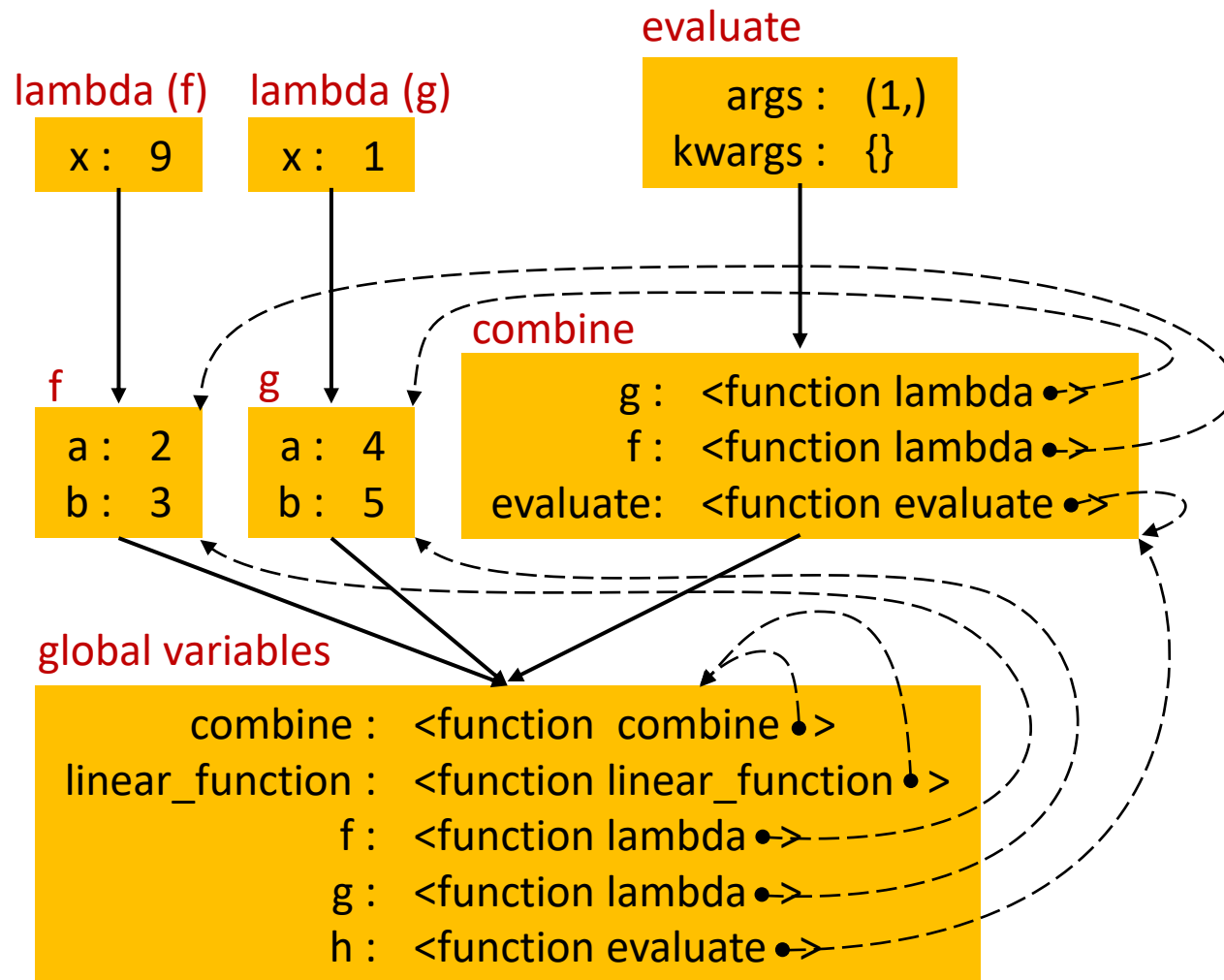


d) 21

e) 25

f) Don't know

Namespace example



linear_combine.py

```
def combine(f, g):  
    def evaluate(*args, **kwargs):  
        return f(g(*args, **kwargs))  
  
    return evaluate  
  
def linear_function(a, b):  
    return lambda x: a * x + b  
  
f = linear_function(2, 3)  
g = linear_function(4, 5)  
  
h = combine(f, g)  
  
print(h(1))
```

partial (trace of computation)

partial_trace.py

```
def partial(fn, *args):
    def new_f(*a):
        print(f'new_f: fn={fn.__name__}, args={args}, a={a}')
        answer = fn(*args, *a)
        print(f'answer={answer}')
        return answer

    return new_f

def f(x, y, z):
    print(f'f({x},{y},{z})')
    return x + 2 * y + 3 * z

g = partial(f, 7)
h = partial(f, 2, 1)
k = partial(g, 1, 2)

print(f'{g(2, 1)=}\n')    # 7 + 2 * 2 + 3 * 1 = 14
print(f'{h(3)=}\n')      # 2 + 2 * 1 + 3 * 3 = 13
print(f'{k()=}\n')        # 7 + 2 * 1 + 3 * 2 = 15
```

Python shell

```
new_f: fn=f, args=(7,), a=(2, 1)
f(7,2,1)
answer=14
g(2, 1)=14

new_f: fn=f, args=(2, 1), a=(3,)
f(2,1,3)
answer=13
h(3)=13

new_f: fn=new_f, args=(1, 2), a=()
new_f: fn=f, args=(7,), a=(1, 2)
f(7,1,2)
answer=15
answer=15
k()=15
```

Python shell

```
> def f(x): return x
> g = lambda x: x
> f.__name__
| 'f'
> g.__name__
| '<lambda>'
```

functools.partial

Object oriented programming

- classes, objects
- self
- construction
- encapsulation

Object Oriented Programming

- **Programming paradigm**, other paradigms are e.g.
 - *functional programming* where the focus is on functions, lambda's and higher order functions, and
 - *imperative programming* focusing on sequences of statements changing the state of the program
- Core concepts are **objects**, **methods** and **classes**,
 - allowing one to construct *abstract data types*, i.e. *user defined types*
 - objects have states
 - methods manipulate objects, defining the interface of the object to the rest of the program
- OO supported by many programming languages, including Python

Object Oriented Programming - History

(selected programming languages)

Mid 1960's **Simula 67**

(Ole-Johan Dahl and Kristen Nygaard, Norsk Regnesentral Oslo)
Introduced classes, objects, virtual procedures

1970's **Smalltalk** (Alan Kay, Dan Ingalls, Adele Goldberg, Xerox PARC)

Object-oriented programming, fully dynamic system
(opposed to the static nature of Simula 67)

1985 **Eiffel** (Bertrand Meyer, Eiffel Software)

Focus on software quality, capturing the full software cycle

1985 **C++** (Bjarne Stroustrup [MSc Aarhus 1975], AT&T Bell Labs)

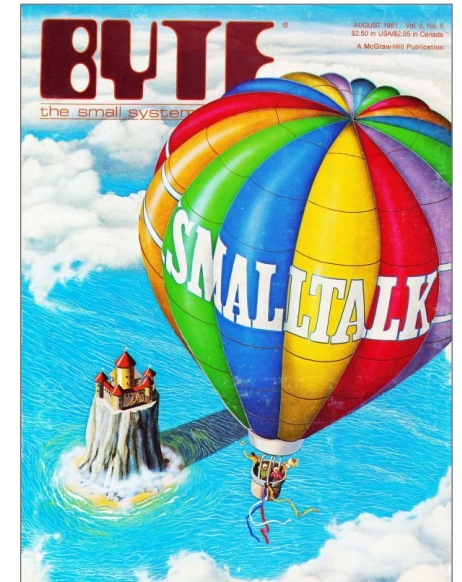
1995 **Java** (James Gosling, Sun)

2000 **C#** (Anders Hejlsberg (studied at DTU) et al., Microsoft)

1991 **Python** (Guido van Rossum)

Multi-paradigm programming language, fully dynamic system

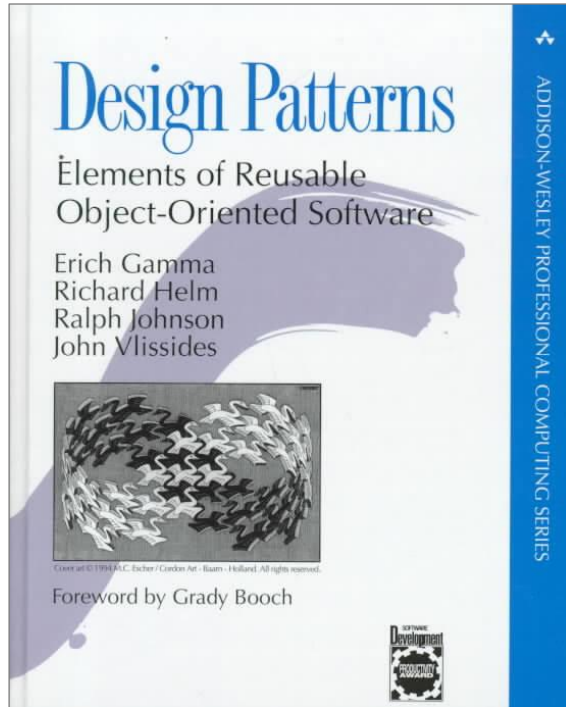
Note: Java, C++, Python, C# are among Top 5 on TIOBE March 2024 index of popular languages (only non OO language among Top 5 was C)



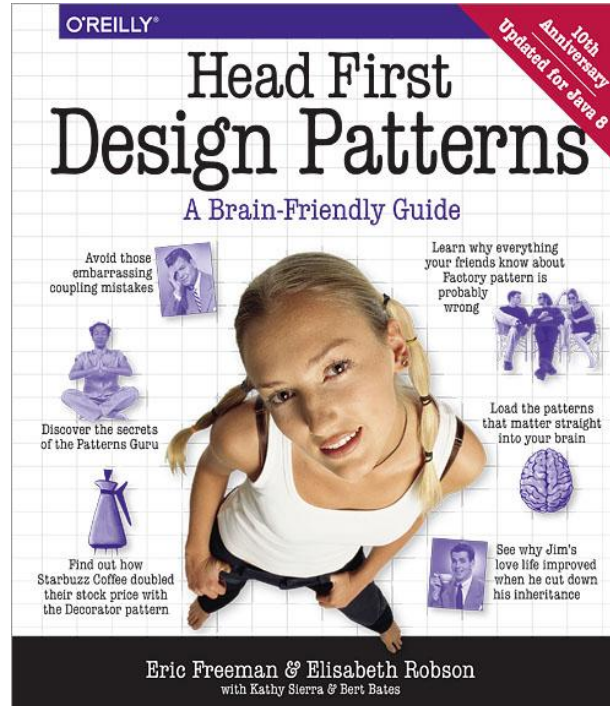
Byte Magazine,
August 1981

Design Patterns (not part of this course)

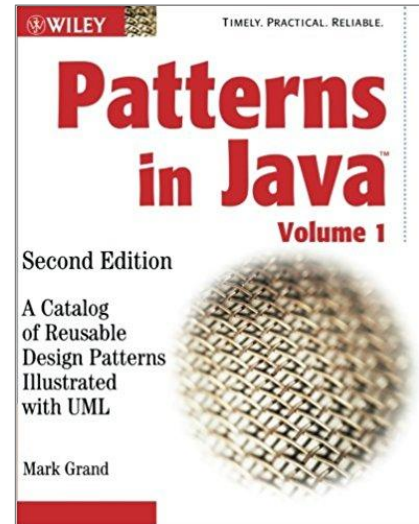
reoccurring patterns in software design



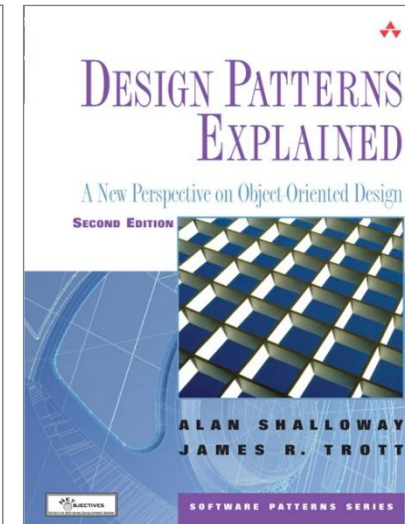
The Classic book 1994
(C++ cookbook)



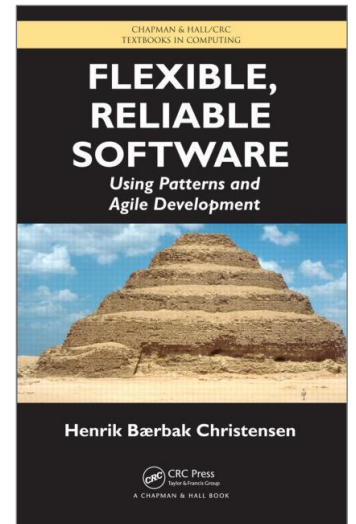
A very alternative book 2004
(Java, very visual)



Java cookbook 2003



Java textbook 2004



Java textbook 2010

...and many more books on the topic of Design Patterns, also with Python

Some known classes, objects, and methods

Type / class	Objects	Methods (examples)
int	0 -7 42 1234567	.__add__(x), .__eq__(x), .__str__()
str	"" 'abc' '12_a'	.isdigit(), .lower(), .__len__()
list	[] [1,2,3] ['a', 'b', 'c']	.append(x), .clear(), .__mul__(x)
dict	{'foo' : 42, 'bar' : 5}	.keys(), .get(), .__getitem__(x)
NoneType	None	.__str__()

Example:

The function `str(obj)` calls the methods
`obj.__str__()` or `obj.__repr__()`, if
`obj.__str__` does not exist.

`print` calls `str`.

Python shell

```
> 5 + 7 # + calls .__add__(7)
| 12
> (5).__add__(7) # eq. to 5 + 7
| 12
> (7).__eq__(7) # eq. to 7 == 7
| True
> 'aBCd'.lower()
| 'abcd'
> 'abcde'.__len__()
| 5
# .__len__() called by len(...)
> ['x', 'y'].__mul__(2)
| ['x', 'y', 'x', 'y']
> {'foo' : 42}.__getitem__('foo')
| 42
# eq. to {'foo' : 42}['foo']
> None.__str__() # used by str(...)
| 'None'
> 'abc'.__str__(), 'abc'.__repr__()
| ('abc', "'abc'")
```


Classes and Objects

class
(type)

class
methods

```
class Student
set_name(name)
set_id(student_id)
get_name()
get_id()
```

objects
(instances)

```
student_DD
name = 'Donald Duck'
id = '107'
```

data
attributes

```
student_MM
name = 'Mickey Mouse'
id = '243'
```

```
student_SM
name = 'Scrooge McDuck'
id = '777'
```

creating **instances**
of class Student
using **constructor**
Student()

Using the Student class

student.py

```
student_DD = Student()
student_MM = Student()
student_SM = Student()

student_DD.set_name('Donald Duck')
student_DD.set_id('107')
student_MM.set_name('Mickey Mouse')
student_MM.set_id('243')
student_SM.set_name('Scrooge McDuck')
student_SM.set_id('777')

students = [student_DD, student_MM, student_SM]

for student in students:
    print(student.get_name(),
          'has id',
          student.get_id())
```

Python shell

```
| Donald Duck has id 107
| Mickey Mouse has id 243
| Scrooge McDuck has id 777
```

Call **constructor** for class Student. Each call returns a new Student object.

Call class methods to set data attributes

Call class methods to read data attributes

class Student

class definitions start with the keyword `class`

often called **mutator methods**, since they change the state of an object

often called **accessor methods**, since they only read the state of an object

class method definitions start with keyword `def` (like normal function definitions)

```
student.py
class Student:
    '''Documentation of class'''

    def set_name(self, name):
        self.name = name

    def set_id(self, student_id):
        self.id = student_id

    def get_name(self):
        return self.name

    def get_id(self):
        return self.id
```

name of class

docstring containing documentation for class

the first argument to all class methods is a reference to the object called upon, and by convention the first argument should be named `self`.

use `self.` to access an attribute of an object or class method (attribute reference)

Note In other OO programming languages the explicit reference to `self` is not required (in Java and C++ `self` is the keyword `this`)

When are object attributes initialized ?

Python shell

```
> x = Student()
> x.set_name("Gladstone Gander")
> x.get_name()
| 'Gladstone Gander'
> x.get_id()
| 
| AttributeError: 'Student' object has no attribute 'id'
```

- Default behaviour of a class is that instances are created with no attributes defined, but has access to the attributes / methods of the class
- In the previous class `Student` both the `name` and `id` attributes were first created when set by `set_name` and `set_id`, respectively

Class construction and `__init__`

- When an object is created using `class_name()` its initializer method `__init__` is called.
- To initialize objects to contain default values, (re)define this function.

student.py

```
class Student:
    def __init__(self):
        self.name = None
        self.id = None

    ... previous method definitions ...
```

Question – What is printed ?

Python shell

```
> class C:
    def __init__(self):
        self.v = 0
    def f(self):
        self.v = self.v + 1
        return self.v

> x = C()
> print(x.f() + x.f())
```

a) 1

b) 2



c) 3

d) 4

e) 5

f) Don't know

__init__ with arguments

- When creating objects using `class_name(args)` the initializer method is called as `__init__(args)`
- To initialize objects to contain default values, (re)define this function to do the appropriate initialization

student.py

```
class Student:
    def __init__(self, name=None, student_id=None):
        self.name = name
        self.id = student_id

... previous method definitions ...
```

Python shell

```
> p = Student('Pluto')
> print(p.get_name())
| Pluto
> print(p.get_id())
| None
```

Are accessor and mutator methods necessary ?

No - but good programming style

Python shell

```
> p = Pair(3, 5)
> p.sum()
| 8
> p.set_a(4)
> p.sum()
| 9
> p.a      # access object attribute
| 4
> p.b = 0  # update object attribute
> p.sum()
| 9      # the_sum not updated
```



pair.py

```
class Pair:
    """ invariant: the_sum = a + b """

    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.the_sum = self.a + self.b

    def set_a(self, a):
        self.a = a
        self.the_sum = self.a + self.b

    def set_b(self, b):
        self.b = b
        self.the_sum = self.a + self.b

    def sum(self):
        return self.the_sum
```


Converting objects to `str`

- To be able to convert an object to a string using `str(object)`, define the method `__str__`
- `__str__` is e.g. used by `print`

Student_constructor.py

```
class Student:
    def __str__(self):
        return f"Student('{self.name}', '{self.id}')"
    ... previous method definitions ...
```

Python shell

```
> print(student_DD)    # without __str__
| <__main__.Student object at 0x03AB6B90>
> print(student_DD)    # with __str__
| Student('Donald Duck', '107')
```

Defining order on instances of a class (sorting)

- To define an order on objects, define the “<” operator by defining `__lt__`
- When “<” is defined a list `L` of students can be sorted using `sorted(L)` and `L.sort()`

student.py

```
class Student:
    def __lt__(self, other):
        return self.id < other.id

    ... previous method definitions ...
```

Python shell

```
> student_DD < student_MM
| True
> [x.id for x in students]
| ['243', '107', '777']
> [x.id for x in sorted(students)]
| ['107', '243', '777']
```

Nothing is private in Python

- Python does not support **hiding information** inside objects
- Recommendation is to start attributes with underscore, if these should be used only locally inside a class, i.e. be considered "private"
- **PEP8**: "Use one leading underscore only for non-public methods and instance variables"

private_attributes.py

```
class My_Class:
    def set_xy(self, x, y):
        self._x = x
        self._y = y

    def get_sum(self):
        return self._x + self._y

obj = My_Class()
obj.set_xy(3, 5)

print('Sum =', obj.get_sum())
print('_x =', obj._x)
```

Python shell

```
| Sum = 8
| _x = 3
```

C++ private, public

C++ vs Python

1. argument types
2. return types
3. `void = NoneType`
4. `private` / `public` access specifier
5. types of data attributes
6. data attributes must be defined in class
7. object creation
8. no `self` in class methods

`private_attributes.cpp`

```
#include <iostream>
using namespace std;

class My_Class {
private: ④
    ⑤int x, y; ⑥
public: ④
    ②③void set_xy(int a, int b) { ①
        x = a;
        y = b;
    }; ⑧
    ②int get_sum() {
        return x + y;
    };
};

main() {
    ⑦My_Class obj;
    obj.set_xy(3, 5);
    cout << "Sum = " << obj.get_sum() << endl;
    cout << "x = " << obj.x << endl;
}
```



invalid reference

Java private, public

Java vs Python

1. argument types
2. return types
3. `void = NoneType`
4. `private` / `public` access specifier
5. types of data attributes
6. data attributes must be defined in class
7. object creation
8. no `self` in class methods

private_attributes.java

```
class My_Class {  
    ④ private ⑤ int x, y; ⑥  
    ④ public ②③ void set_xy(int a, int b) { ①  
        x = a; y = b;  
    }  
    ④ public ② int get_sum() ⑧ { return x + y; };  
};  
  
class private_attributes {  
    public static void main(String args[]){  
        ⑦ My_Class obj = new My_Class();  
        obj.set_xy(3, 5);  
        System.out.println("Sum = " + obj.get_sum());  
        System.out.println("x = " + obj.x);  
    }  
}
```

invalid reference 

Name mangling (partial privacy)

- Python handles references to class attributes inside a class definition with *at least two leading underscores and at most one trailing underscore* in a special way: `__attribute` is textually replaced by `__classname__attribute`
- Note that [Guttag, p. 200] states “that attribute is not visible outside the class” – which only is partially correct (see example)

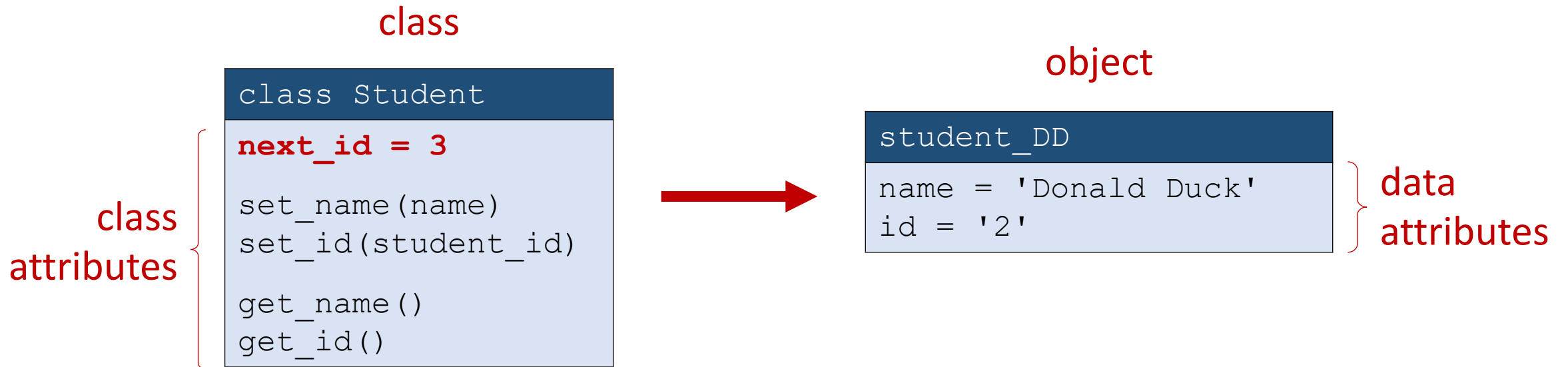
name_mangling.py

```
class MySecretBox:
    def __init__(self, secret):
        self.__secret = secret
```

Python shell

```
> x = MySecretBox(42)
> print(x.__secret)
| AttributeError: 'MySecretBox'
  object has no attribute
    '__secret'
> print(x.__MySecretBox__secret)
| 42
```

Class attributes



- `obj.attribute` first searches the objects attributes to find a match, if no match, continuous to search the attributes of the class
- Assignments to `obj.attribute` are always to the objects attribute (possibly creating the attribute)
- Class attributes can be accessed directly as `class.attribute` (or `obj.__class__.attribute`)

Class data attribute

- `next_id` is a class attribute
- Accessed using `Student.next_id`
- The lookup ① can be replaced with `self.next_id`, since only the class has this attribute, looking up in the object will be propagated to a lookup in the class attributes
- In the update ② it is crucial that we update the class attribute, since otherwise the incremented value will be assigned as an object attribute (What will the result be?)

student_auto_id.py

```
class Student:
    next_id = 1 # class attribute
    def __init__(self, name):
        self.name = name
        self.id = str(Student.next_id) ①
        ② Student.next_id += 1
    def get_name(self):
        return self.name
    def get_id(self):
        return self.id

students = [Student('Scrooge McDuck'),
            Student('Donald Duck'),
            Student('Mickey Mouse')]

for student in students:
    print(student.get_name(),
          "has student id",
          student.get_id())
```

Python shell

```
| Scrooge McDuck has student id 1
| Donald Duck has student id 2
| Mickey Mouse has student id 3
```


Question – What does `obj.get()` return ?

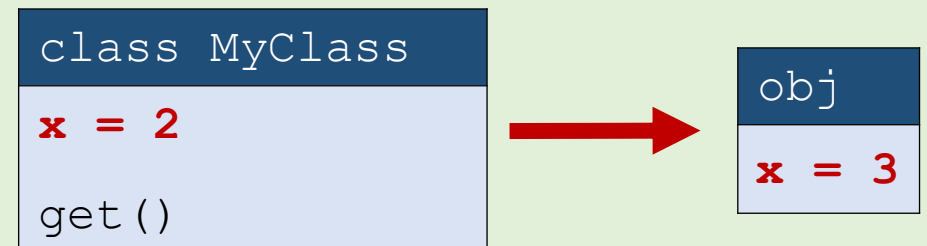
Python shell

```
> class MyClass:
    x = 2

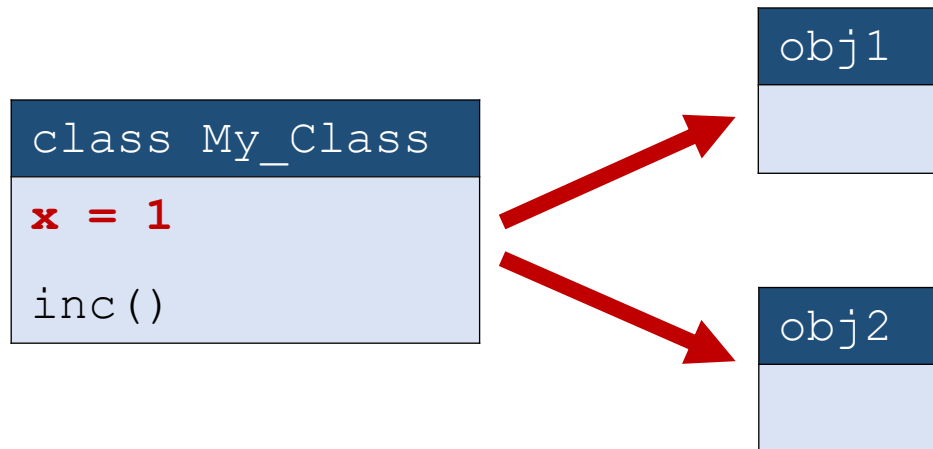
    def get(self):
        self.x = self.x + 1
        return MyClass.x + self.x

> obj = MyClass()
> print(obj.get())
| ?
```

- a) 4
- 😊 b) 5
- c) 6
- d) UnboundLocalError
- e) Don't know



Class data attribute example (in Python)



```
class_attributes.py
class My_Class:
    x = 1  # class attribute
    def inc(self):
        My_Class.x = self.x + 1
obj1 = My_Class()
obj2 = My_Class()
obj1.inc()
obj2.inc()
print(obj1.x, obj2.x)
```

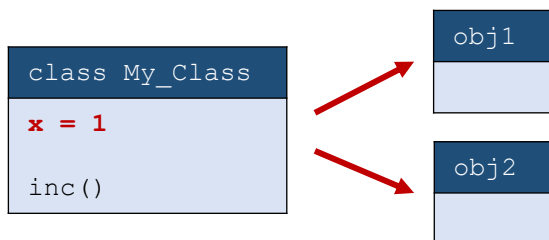
Python shell

| 3 3

- Note that `My_Class.x` and `self.x` refer to the same class attribute (since `self.x` has never been assigned a value)

Java static

- In Java *class attributes*, i.e. attribute values shared by all instances, are labeled **static**
- Python allows both class and instance attributes with the same name – in Java at most one of them can exist



static_attributes.java

```
class My_Class {
    public static int x = 1;
    public void inc() { x += 1; };
}

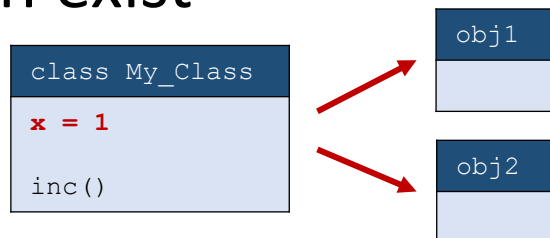
class static_attributes {
    public static void main(String args[]){
        My_Class obj1 = new My_Class();
        My_Class obj2 = new My_Class();
        obj1.inc();
        obj2.inc();
        System.out.println(obj1.x);
        System.out.println(obj2.x);
    }
}
```

Java output

```
| 3
| 3
```

C++ static

- In C++ *class attributes*, i.e. attribute values shared by all instances, are labeled **static**
- ISO C++ forbids in-class initialization of non-const static member
- Python allows both class and instance attributes with the same name – in C++ at most one of them can exist



static_attributes.cpp

```
#include <iostream>
using namespace std;

class My_Class {
public:
    static int x; // "= 1" is not allowed
    void inc() { x += 1; };
};

int My_Class::x = 1; // class initialization

int main() {
    My_Class obj1;
    My_Class obj2;
    obj1.inc();
    obj2.inc();
    cout << obj1.x << endl;
    cout << obj2.x << endl;
}
```

C++ output

```
| 3
| 3
```

Constants

- A simple usage of class data attributes is to store a set of constants (but there is nothing preventing anyone to change these values)

Python shell

```
> class Color:
    RED    = "ff0000"
    GREEN  = "00ff00"
    BLUE   = "0000ff"
> Color.RED
| 'ff0000'
```

PEP8 Style Guide for Python Code (some quotes)

- Class names should normally use the **CapWords** convention.
- Always use **self** for the first argument to instance methods.
- Use one **leading underscore** only for **non-public methods and instance variables**.
- For **simple public data attributes**, it is best to expose just the attribute name, **without complicated accessor/mutator methods**.
- Always decide whether a class's methods and instance variables (collectively "attributes") should be **public** or **non-public**.
If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

Some methods many classes have

Method	Description
<code>__eq__(self, other)</code>	Used to test if two elements are equal Two elements where <code>__eq__</code> is true must have equal <code>__hash__</code>
<code>__str__(self)</code>	Used by <code>str</code> and <code>print</code>
<code>__repr__(self)</code>	Used by <code>repr</code> , e.g. for printing to the IDE shell (usually something that is a valid Python expression for <code>eval()</code>)
<code>__len__(self)</code>	Length (integer) of object, e.g. lists, strings, tuples, sets, dictionaries
<code>__doc__</code>	The docstring of the class
<code>__hash__(self)</code>	Returns hash value (integer) of object Dictionary keys and set values must have a <code>__hash__</code> method
<code>__lt__(self, other)</code>	Comparison (less than, <code><</code>) used by <code>sorted</code> and <code>sort()</code>
<code>__init__(self, ...)</code>	Class initializer

Class hierarcies

- inheritance
- method overriding
- super
- multiple inheritance

Calling methods of a class

- If an object *obj* of class *C* has a method *method*, then usually you call *obj.method()*
- It is possible to call the method in the class directly using *C.method*, where the object is the first argument

C.method(obj)

x.py

```
class X:
    def set_x(self, x):
        self.x = x

    def get_x(self):
        return self.x

obj = X()

obj.set_x(42)

print(f'{obj.get_x()} = {obj.x}')
print(f'{obj.x = }')
print(f'{X.get_x(obj) = }')
```

Python shell

```
| obj.get_x() = 42
| obj.x = 42
| X.get_x(obj) = 42
```

Classes and Objects

Observation: **students** and **employees** are **persons** with additional attributes

```
class Person
```

```
    set_name(name)
    get_name()
    set_address(address)
    get_address()
```

instance



```
Person object
```

```
name = 'Mickey Mouse'
address = 'Mouse Street 42, Duckburg'
```

```
class Student
```

```
    set_name(name)
    get_name()
    set_address(address)
    get_address()
    set_id(student_id)
    get_id()
    set_grade(course, grade)
    get_grades()
```

instance



```
Student object
```

```
name = 'Donald Duck'
address = 'Duck Steet 13, Duckburg'
id = '1094'
grades = {'programming' : 'A' }
```

```
Employee object
```

```
name = 'Goofy'
address = 'Clumsy Road 7, Duckburg'
employer = 'Yarvard University'
```

Classes and Objects

```
class Person
    set_name(name)
    get_name()
    set_address(address)
    get_address()
```

```
class Student
    set_name(name)
    get_name()
    set_address(address)
    get_address()
    set_id(student_id)
    get_id()
    set_grade(course, grade)
    get_grades()
```

person attributes

Goal – avoid redefining the 4 methods below from person class again in student class

person.py

```
class Person:
    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_address(self, address):
        self.address = address

    def get_address(self):
        return self.address
```

Classes inheritance

```
class Person
    set_name(name)
    get_name()
    set_address(address)
    get_address()
```

```
class Student
    set_name(name)
    get_name()
    set_address(address)
    get_address()
    set_id(student_id)
    get_id()
    set_grade(course, grade)
    get_grades()
```

person attributes

class Student **inherits** from class Person
class Person is the **base class** of Student

person.py

```
class Student(Person):
    def set_id(self, student_id):
        self.id = student_id

    def get_id(self):
        return self.id

    def set_grade(self, course, grade):
        self.grades[course] = grade

    def get_grades(self):
        return self.grades
```

Classes constructors

```
class Person
```

```
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
```

```
set_name(name)
get_name()
set_address(address)
get_address()

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

person
attributes

```
person.py
```

```
class Person:
    def __init__(self):
        self.name = None
        self.address = None
    ...
```

} constructor for
Person class

```
class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
    ...
```

} constructor for
Student class

Notes

- 1) If `Student.__init__` is not defined, then `Person.__init__` will be called
- 2) `Student.__init__` must call `Person.__init__` to initialize the name and address attributes

super()

```
class Person
```

```
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
```

```
set_name(name)
get_name()
set_address(address)
get_address()
set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()
```

person
attributes

```
person.py
```

```
class Person:
    def __init__(self):
        self.name = None
        self.address = None
        ...

class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
        super().__init__()
        ...
```

} alternative
constructor

Notes

- 1) Function `super()` searches for attributes in base class
- 2) `super` is often a keyword in other OO languages, like Java and C++
- 3) Note `super().__init__()` does not need `self` as argument

Method search order

```
class Person
set_name(name)
get_name()
set_address(address)
get_address()
```



parent class

```
class Student(Person)
set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()
```

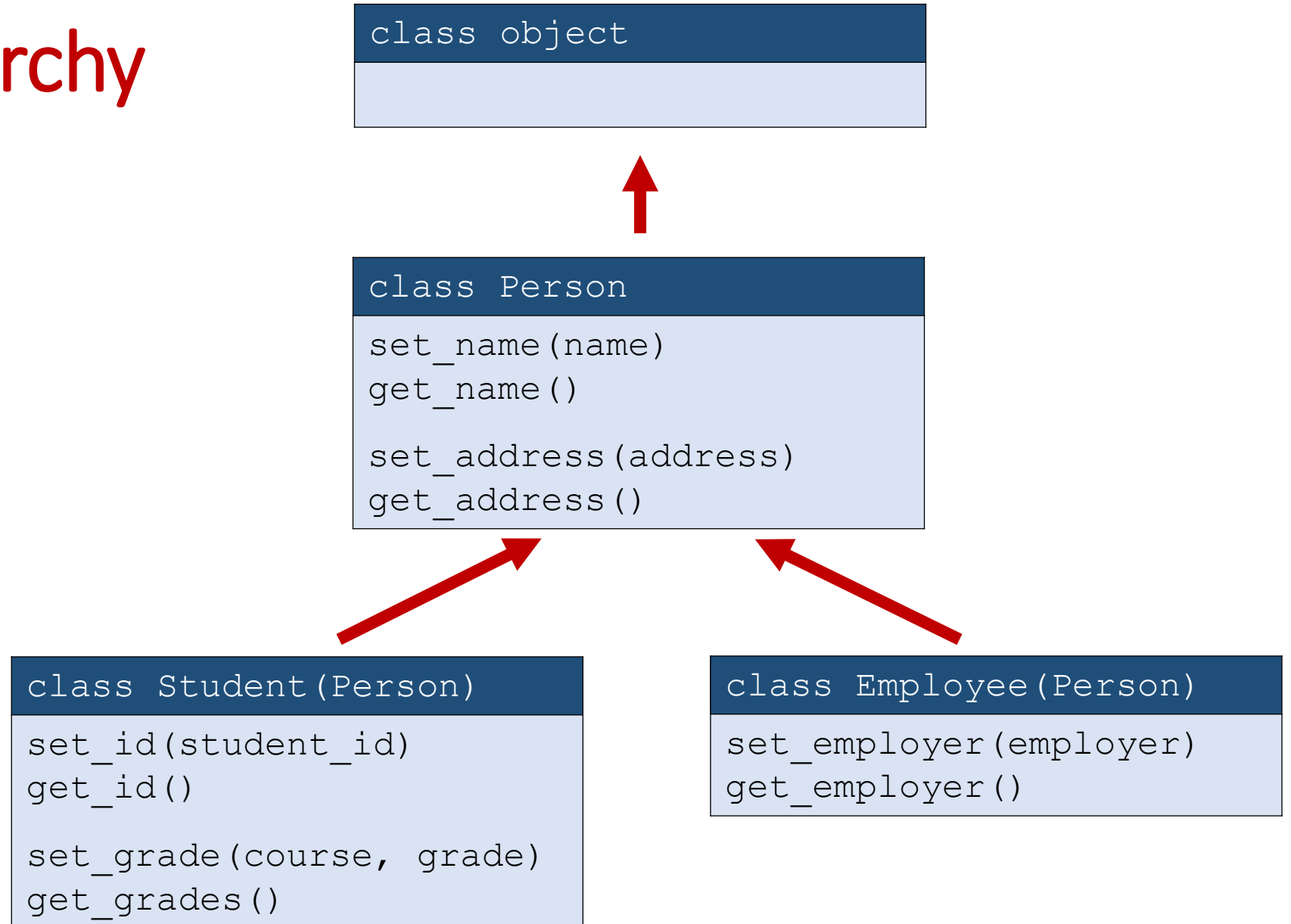
instance of



Student object

```
name = 'Donald Duck'
address = 'Duck Steet 13, Duckburg'
id = '1094'
grades = {'programming' : 'A' }
```


Class hierarchy



Method overriding

overloading.py

```
class A:
    def say(self):
        print('A says hello')

class B(A): # B is a subclass of A
    def say(self):
        print('B says hello')
        super().say()
```

Python shell

```
> B().say()
| B says hello
| A says hello
```

class A
say()



class B
say()

In Java one can use the keyword "finally" to prevent any subclass to override a method

Question – What does `b.f()` print ?

Python shell

```
> class A:
    def f(self):
        print("Af")
        self.g()
    def g(self):
        print("Ag")

> class B(A):
    def g(self):
        print("Bg")

> b = B()
> b.f()
| ?
```

a) AttributeError

b) Af Ag

 c) Af Bg

d) Don't know

Undefind methods in superclass ?

Python shell

```
> class A:
    def f(self):
        print("Af")
        self.g()

    def g(self):
        print("Ag")

> class B(A):
    def g(self):
        print("Bg")

> b = B()
> b.f()
| Af
| Bg

> a = A()
> a.f()
| Af
| Ag
```

Python shell

```
> class A:
    def f(self):
        print("Af")
        self.g()

> class B(A):
    def g(self):
        print("Bg")

> b = B()
> b.f()
| Af
| Bg

> a = A()
> a.f()
| Af
| AttributeError: 'A' object has no attribute 'g'
```

method `g` undefined in class `A`;
subclasses must implement `g`
to be able to call `f`

in Java, `A` would have been
required to be declared an
abstract class

can create instance of `A`

fails since `g` is not
defined in class `A`

Name mangling and inheritance



Python shell

```
> class A:
    def f(self):
        print("Af")
        self.__g()
    def __g(self):
        print("Ag")
> class B(A):
    def __g(self):
        print("Bg")
> b = B()
> b.f()
| Af
| Ag
```

- The call to `A.__g` in `A.f` forces a call to `__g` to stay within `A`
- Recall that due to name mangling, `__g` is accessible as `A._A__g`

Multiple inheritance

- A class can inherit attributes from multiple classes (in example two)
- When calling a method defined in several ancestor classes, Python executes only one of these (in the example `say_hello`)
- Which one is determined by the so called "C3 Method Resolution Order" (originating from the Dylan language)

Raymond Hettinger, *Super considered super!*
Conference talk at PyCon 2015

multiple_inheritance.py

```
class Alice:
    def say_hello(self):
        print("Alice says hello")
    def say_good_night(self):
        print("Alice says good night")
class Bob:
    def say_hello(self):
        print("Bob says hello")
    def say_good_morning(self):
        print("Bob says good morning")
class X(Alice, Bob): # Multiple inheritance
    def say(self):
        self.say_good_morning()
        self.say_hello() # C3 resolution
        Alice.say_hello(self) # from Alice
        Bob.say_hello(self) # from Bob
        self.say_good_night()
```

Python shell

```
> X().say()
| Bob says good morning
| Alice says hello ← since Alice before Bob
| Alice says hello   in list of super classes
| Bob says hello
| Alice says good night
```

C3 Method resolution order

- Use `help(class)` to determine the resolution order for the class
- or access the `__mro__` attribute of the class

Python shell

```
> X.__mro__
| (<class '__main__.X'>, <class '__main__.Alice'>,
| <class '__main__.Bob'>, <class 'object'>)
> help(X)
| Help on class X in module __main__:
| class X(Alice, Bob)
|     Method resolution order:
|         X
|         Alice
|         Bob
|         builtins.object
|     Methods defined here:
|         say(self)
|         -----
|     Methods inherited from Alice:
|         say_good_night(self)
|         say_hello(self)
|         -----
|         ...
|         -----
|     Methods inherited from Bob:
|         say_good_morning(self)
```

Question – Who says hello ? Bob says good morning

inheritance.py

```
class Alice:
    def say_hello(self):
        print("Alice says hello")
class Bob:
    def say_hello(self):
        print("Bob says hello")
    def say_good_morning(self):
        self.say_hello()
        print("Bob says good morning")
class X(Alice, Bob): # Multiple inheritance
    pass

X().say_good_morning()
```



- a) Alice
- b) Bob
- c) Dont' know

...example of code injection using multiple inheritance and where body of new class is empty

Comparing objects and classes

- `id(obj)` returns a unique identifier for an object (in CPython the memory address)
- `obj1 is obj2` tests if `id(obj1) == id(obj2)`
- `type(obj)` and `obj.__class__` return the class of an object
- `isinstance(object, class)` checks if an object is of a particular class, *or a derived subclass*
- `issubclass(class1, class2)` checks if `class1` is a subclass of `class2`

Note: PEP8 recommends to use `isinstance(x, int)` over `type(x) is int`

`is` is not for integers, strings, ... and `is` is not `==`

Python shell



```
> 500 + 500 is 1000
| True
> x = 500
> x + x is 1000
| False
> x + x == 1000 # int.__eq__(...)
| True
> for x in range(0, 1000):
    if x - 1 + 1 is not x:
        print(x)
        break
| 257
> for x in range(0, -1000, -1):
    if x + 1 - 1 is not x:
        print(x)
        break
| -6
```



Python shell



```
> "abc" is "abc"
| True
> "abc" is "xabc"[1:]
| False
> x, y = "abc", "xabc"[1:]
> x, y
| ('abc', 'abc')
> x is y
| False
> x == y # x.__eq__(y)
| True
```



- Only use `is` on objects !
- Even though `isinstance(42, object)` and `isinstance("abc", object)` are true, do not use `is` on integers and strings !

Comparison of OO in Python, Java and C++

- private, public, – in Python everything in an object is public
- class inheritance – core concept in OO programming
 - Python and C++ support multiple inheritance
 - Java only allows single inheritance, but Java "interfaces" allow for something like multiple inheritance
- Python and C++ allow overloading standard operators (+, *, ...).
In Java it is not possible
- Overloading methods
 - Python extremely dynamic (hard to say anything about the behaviour of a program in general)
 - Java and C++'s type systems allow several methods with same name in a class, where they are distinguished by the type of the arguments, whereas Python allows only one method that can have * and ** arguments

Python is really dynamic...

(this is ugly – likely don't do this at home)

Python shell

```
> class Pair:
    def __init__(self, x, y):
        self._x = x
        self._y = y

> point = Pair(3, 5)
> print(point) # class Pair has no __str__ method, uses object.__str__
| <__main__.Pair object at 0x0000027571904B50>

> Pair.__str__ = lambda self: f'Pair({self._x}, {self._y})'

> print(point)
| Pair(3, 5)
```



← dynamically add a method to an existing class (and all existing instances),
e.g. technique used by the class decorator `@functools.total_ordering`

C++ example

- Multiple methods with identical name (`print`)
- The types distinguish the different methods

printing.py

```
class MyClass:
    def print(self, value):
        if isinstance(value, int):
            print('An integer', value)
        elif isinstance(value, str):
            print('A string', value)

C = MyClass()
C.print(42)
C.print('abc')
```

printing.cpp

```
#include <iostream>
using namespace std;

class MyClass {
public:
    void print(int x) {
        cout << "An integer " << x << endl;
    };
    void print(string s) {
        cout << "A string " << s << endl;
    };
};

main() {
    MyClass C;
    C.print(42);
    C.print("abc");
}
```

Shell

```
| An integer 42
| A string abc
```

Exceptions and file input/output

- try-raise-except-finally
- Exception
- control flow
- match - case
- file open/read/write
- sys.stdin, sys.stdout, sys.stderr
- context manager

Exceptions – Error handling and control flow

- **Exceptions** is a widespread technique to handle run-time **errors** / abnormal behaviour (e.g. in Python, Java, C++, JavaScript, C#)
- **Exceptions** can also be used as an **advanced control flow mechanism** (e.g. in Python, Java, JavaScript)
 - *Problem: How to perform a "break" in a recursive function ?*

Built-in exceptions (class hierarchy)

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |   +-- FloatingPointError
        |   +-- OverflowError
        |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |   +-- ModuleNotFoundError
    +-- LookupError
        |   +-- IndexError
        |   +-- KeyError
    +-- MemoryError
    +-- NameError
        |   +-- UnboundLocalError
    +-- TypeError
    +-- ValueError
        |   +-- UnicodeError
        |       +-- UnicodeDecodeError
        |       +-- UnicodeEncodeError
        |       +-- UnicodeTranslateError
```

```
+-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |       |   +-- BrokenPipeError
    |       |   +-- ConnectionAbortedError
    |       |   +-- ConnectionRefusedError
    |       |   +-- ConnectionResetError
    +-- FileNotFoundError
    +-- InterruptedError
    +-- IsADirectoryError
    +-- NotADirectoryError
    +-- PermissionError
    +-- ProcessLookupError
    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
    |   +-- NotImplementedError
    |   +-- RecursionError
+-- SyntaxError
    |   +-- IndentationError
    |   +-- TabError
+-- SystemError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```


Typical built-in exceptions

and unhandled
behaviour

Python shell

```
> 7 / 0
| ZeroDivisionError: division by zero
> int('42x')
| ValueError: invalid literal for int() with base 10: '42x'
> x = y
| NameError: name 'y' is not defined
> L = [1] * 10_000_000_000
| MemoryError
> 2.5 ** 1000
| OverflowError: (34, 'Result too large')
> t = (3, 4)
> t[0] = 7
| TypeError: 'tuple' object does not support item assignment
> t[3]
| IndexError: tuple index out of range
> t.x
| AttributeError: 'tuple' object has no attribute 'x'
> x = {}
> x['foo']
| KeyError: 'foo'
> def f(x): f(x + 1)
> f(0)
| RecursionError: maximum recursion depth exceeded
> def f(): x = x + 1
> f()
| UnboundLocalError: local variable 'x' referenced before assignment
```

Catching exceptions – Fractions (I)

fraction1.py

```
while True:
    numerator = int(input('Numerator = '))
    denominator = int(input('Denominator = '))
    result = numerator / denominator
    print(f'{numerator} / {denominator} = {result}')
```

Python shell

```
| Numerator = 10
| Denominator = 3
| 10 / 3 = 3.3333333333333335
| Numerator = 20
| Denominator = 0
| ZeroDivisionError: division by zero
```

>

Catching exceptions – Fractions (II)

fraction2.py

```
while True:
    numerator = int(input('Numerator = '))
    denominator = int(input('Denominator = '))
    try:
        result = numerator / denominator
    except ZeroDivisionError:
        print('cannot divide by zero')
        continue
    print(f'{numerator} / {denominator} = {result}')
```

catch
exception

Python shell

```
| Numerator = 10
| Denominator = 0
| cannot divide by zero
| Numerator = 20
| Denominator = 3
| 20 / 3 = 6.666666666666667
| Numerator = 42x
| ValueError: invalid literal for int() with base 10: '42x'
```

Catching exceptions – Fractions (III)

fraction3.py

```
while True:
    try:
        numerator = int(input('Numerator = '))
        denominator = int(input('Denominator = '))
    except ValueError:
        print('input not a valid integer')
        continue
    try:
        result = numerator / denominator
    except ZeroDivisionError:
        print('cannot divide by zero')
        continue
    print(f'{numerator} / {denominator} = {result}')
```

catch
exception

catch
exception

Python shell

```
| Numerator = 5
| Denominator = 2x
| input not a valid integer
| Numerator = 5
| Denominator = 2
| 5 / 2 = 2.5
```

fraction3.py

```
while True:
    try:
        numerator = int(input('Numerator = '))
        denominator = int(input('Denominator = '))
    except ValueError:
        print('input not a valid integer')
        continue
    try:
        result = numerator / denominator
        print(f'{numerator} / {denominator} = {result}')
    except ZeroDivisionError:
        print('cannot divide by zero')
```

```
Python shell
```

[illegible]

exception
not caught

Catching exceptions – Fractions (IV)

catch
exceptions

fraction4.py

```
while True:
    try:
        numerator = int(input('Numerator = '))
        denominator = int(input('Denominator = '))
        result = numerator / denominator
        print(f'{numerator} / {denominator} = {result}')
    except ValueError:
        print('input not a valid integer')
    except ZeroDivisionError:
        print('cannot divide by zero')
```

Python shell

```
| Numerator = 3
| Denominator = 0
| cannot divide by zero
| Numerator = 3x
| input not a valid integer
| Numerator = 4
| Denominator = 2
| 4 / 2 = 2.0
```

Keyboard interrupt (*Ctrl-c*)

- throws **KeyboardInterrupt** exception

infinite-loop1.py

```
print('starting infinite loop')

x = 0
while True:
    x = x + 1

print(f'done ({x = })')
input('type enter to exit')
```

Python shell

```
| starting infinite loop
| Traceback (most recent call last):
|   File 'infinite-loop1.py', line 4, in <module>
|     x = x + 1
| KeyboardInterrupt
```

infinite-loop2.py

```
print('starting infinite loop')

try:
    x = 0
    while True:
        x = x + 1
except KeyboardInterrupt:
    pass

print(f'done ({x = })')
input('type enter to exit')
```

Python shell

```
| starting infinite loop
| done (x = 23890363)      Ctrl-c
| type enter to exit
```

Keyboard interrupt (*Ctrl-c*)

- Be aware that you likely would like to leave the *Ctrl-c* generated **KeyboardInterrupt** exception unhandled, except when stated explicitly

read-int1.py

```
while True:
    try:
        x = int(input('An integer: '))
        break
    except ValueError: # only ValueError
        continue
```

```
print('The value is:', x)
```

Python shell

```
| An integer:           Ctrl-c
| KeyboardInterrupt
```

read-int2.py

```
while True:
    try:
        x = int(input('An integer: '))
        break
    except: # all exceptions
        continue
```

```
print('The value is:', x)
```

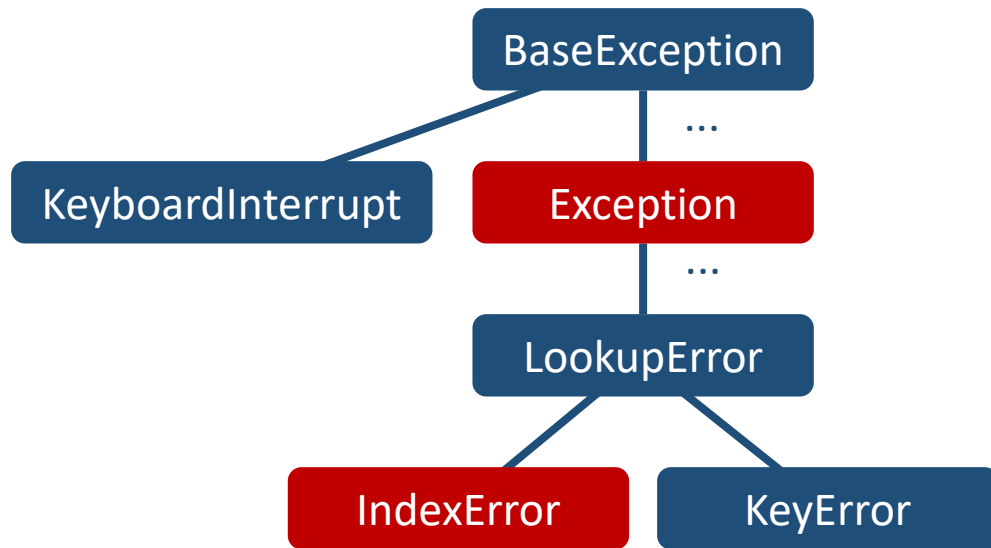
Python shell

```
| An integer:           Ctrl-c
| An integer:           Ctrl-c
| An integer:
```

catches
KeyboardInterrupt

- (left) KeyboardInterrupt is unhandled (right) it is handled (intentionally?)

Exception class hierarchy



except-twice1.py

```
try:
    L[4]
except IndexError: # must be before Exception
    print('IndexError')
except Exception:
    print('Fall back exception handler')
```

except-twice2.py

```
try:
    L[4]
except Exception: # and subclasses of Exception
    print('Fall back exception handler')
except IndexError:
    print('IndexError') # unreachable
```



try statement syntax

try:

code

except ExceptionType1:

code # executed if raised exception instanceof
ExceptionType1 (or subclass of ExceptionType1)

except ExceptionType2:

code # executed if exception type matches and none of
the previous except statements matched

...

else:

code # only executed if no exception was raised

finally:

code # always executed independent of exceptions
typically used to clean up (like closing files)

arbitrary number of except cases

except variations

`except:` # catch all exceptions



`except ExceptionType:` # only catch exceptions of class ExceptionType
or subclasses of ExceptionType

`except (ExceptionType1, ExceptionType2, ..., ExceptionTypek):`
catch any of k classes (and subclasses)
parenthesis cannot be omitted

`except ExceptionType as e:`
catch exception and assign exception object to e
e.args contains arguments to the raised exception

Raising exceptions

- An exception is raised (or thrown) using one of the following (the first being an alias for the second):

```
raise ExceptionType
```

```
raise ExceptionType()
```

```
raise ExceptionType(args)
```

abstract.py

```
class A():
    def f(self):
        print('f')
        self.g()

    def g(self):
        raise NotImplementedError

class B(A):
    def g(self):
        print('g')
```

Python shell

```
> B().f()
| f
| g
> A().f()
| f
| NotImplementedError
```

User exceptions

- New exception types are created using **class inheritance** from an existing exception type (possibly defining `__init__`)

tree-search.py

```
class SolutionFound(Exception): # new exception
    pass

def recursive_tree_search(x, tree):
    if isinstance(tree, tuple):
        for child in tree:
            recursive_tree_search(x, child)
    elif x == tree:
        raise SolutionFound # found x in tree

def tree_search(x, tree):
    try:
        recursive_tree_search(x, tree)
    except SolutionFound:
        print('found', x)
    else:
        print('search for', x, 'unsuccessful')
```

Python shell

```
> tree_search(8, ((3,2),5,(7,(4,6))))
| search for 8 unsuccessful
> tree_search(7, ((3,2),5,(7,(4,6))))
| found 7
```

PEP8 on exceptions

- For all try/except clauses, limit the try clause to the absolute **minimum amount of code** necessary
- The class naming convention applies (**CapWords**)
- Use the **suffix "Error"** on your exception names (if the exception actually is an error)
- A bare **except:** clause will catch `SystemExit` and `KeyboardInterrupt` exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use **except Exception:**

3 ways to read lines from a file

Steps

1. Open file using `open`
2. Read lines from file using
 - a) `for line in filehandler:`
 - b) `filehandler.readlines`
 - c) `filehandler.readline`
3. Close file using `close`

`open('filename.txt')` assumes the file to be in the same folder as your Python program, but you can also provide a full path
`open('c:/Users/gerth/Documents/filename.txt')`

try to open file for reading
filename
filehandle
iterate over lines in file
close file when done

```
reading-file1.py
f = open('reading-file1.py')
for line in f:
    print('> ', line, end='')
f.close()
```

read all lines into a list of strings

```
reading-file2.py
f = open('reading-file2.py')
lines = f.readlines()
f.close()
for line in lines:
    print('> ', line, end='')
```

read single line (terminated by '\n')

```
reading-file3.py
f = open('reading-file3.py')
line = f.readline()
while line != '':
    print('> ', line, end='')
    line = f.readline()
f.close()
```

3 ways to write lines to a file

- Opening file:
`open(filename, mode)`
where *mode* is a string, either 'w' for opening a new (or truncating an existing file) and 'a' for appending to an existing file
- Write single string:
`filehandle.write(string)`
Returns the number of characters written
- Write list of strings strings:
`filehandle.writeline(list)`
- Newlines ('\\n') must be written explicitly
- `print` can take an optional `file` argument

try to open file
for writing

write mode

write single
string to file

write list of
strings to file

append to existing file

```
write-file.py
f = open('output-file.txt', 'w')
f.write('Text 1\\n')
f.writelines(['Text 2\\n', 'Text 3 '])
f.close()

g = open('output-file.txt', 'a')
print('Text 4', file=g)
g.writelines(['Text 5 ', 'Text 6'])
g.close()
```

output-file.txt

```
Text 1
Text 2
Text 3 Text 4
Text 5 Text 6
```


Exceptions while dealing with files

- When dealing with files one should be prepared to handle errors / raised exceptions, e.g. `FileNotFoundError`

reading-file4.py

```
try:
    f = open('reading-file4.py')
except FileNotFoundError:
    print('Could not open file')
else:
    try:
        for line in f:
            print('> ', line, end='')
    finally:
        f.close()
```

Opening files using `with` (recommended way)

- The Python keyword `with` allows to create a *context manager* for handling files
- Filehandle will automatically be closed, also when exceptions occur
- Under the hood: filehandles returned by `open` support `__enter__` and `__exit__` methods

`f` = result of calling `__enter__()`
on result of `open` expression,
which is the file handle

reading-file5.py

```
with open('reading-file5.py') as f:
    for line in f:
        print('> ', line, end='')
```

Does a file exist?

- Module `os.path` contains a method `isfile` to check if a file exists

checking-files.py

```
import os.path

filename = input('Filename: ')
if os.path.isfile(filename):
    print('file exists')
else:
    print('file does not exist')
```

module `sys`

- Module `sys` contains the three standard file handles
`sys.stdin` (used by the `input` function)
`sys.stdout` (used by the `print` function)
`sys.stderr` (error output from the Python interpreter)

`sys-test.py`

```
import sys
sys.stdout.write('Input an integer: ')
x = int(sys.stdin.readline())
sys.stdout.write(f'{x} square is {x ** 2}')
```

Python shell

```
| Input an integer: 10
| 10 square is 100
```

`print(..., file=output file)`

`sys-print-file.py`

```
import sys
def complicated_function(file):
    print('Hello world', file=file) # print to file or STDOUT
while True:
    file_name = input('Output file (empty for STDOUT): ')

    if file_name == '':
        file = sys.stdout
        break
    else:
        try:
            file = open(file_name, 'w')
            break
        except Exception:
            pass

complicated_function(file)
if file != sys.stdout:
    file.close()
```

Input performance

- Reading 10.000.000 short lines

<code>input()</code>	2.2 sec
<code>sys.stdin.readlines()</code>	0.65 sec
<code>sys.stdin.readline()</code>	0.65 sec
<code>readline()</code> alias	0.50 sec
<code>for line in sys.stdin:</code>	0.39 sec
<code>sys.stdin.read()</code>	0.18 sec
<code>sys.stdin.read().split('\n')</code>	0.53 sec

- Note: Time difference up to a factor 10

input_performance.py

```
from time import time
import sys

n = 10_000_000
data = 'numbers.txt'

def test_input():
    for i in range(n):
        line = input()

def test_stdin_readline():
    for i in range(n):
        line = sys.stdin.readline()

def test_stdin_readline():
    readline = sys.stdin.readline
    for i in range(n):
        line = readline()

def test_stdin_readlines():
    lines = sys.stdin.readlines()

def test_stdin_read():
    text = sys.stdin.read()

def test_for():
    for line in sys.stdin:
        pass

def create_test_input():
    with open(data, 'w') as file:
        for i in range(n):
            print(i, file=file)

create_test_input()
for _ in range(5):
    for name, value in list(vars().items()):
        if name.startswith('test_'):
            with open(data) as file:
                original_stdin = sys.stdin
                sys.stdin = file
                start = time()
                value()
                end = time()
                sys.stdin = original_stdin
            print(f'{name} {end - start:.2} sec')
```

Performance of scanning a file

- Python can efficiently scan through quite big files

File	Size	Time
<u>Atom_chem_shift.csv</u>	≈ 750 MB	≈ 8 sec
<u>cano.txt</u>	≈ 3.7 MB	≈ 0.1 sec

The first search finds all lines related to ThrB12-DKP-insulin (Entry ID 6203) in a chemical database available from www.bmrb.wisc.edu

The second search finds all occurrences of “Germany” in Conan Doyle's complete Sherlock Holmes available at sherlock-holm.es

file-scanning.py

```
from time import time
for filename, query in [
    ('Atom_chem_shift.csv', '6203'),
    ('cano.txt', 'Germany')
]:
    count = 0
    matches = []
    start = time()
    with open(filename) as f:
        for i, line in enumerate(f, start=1):
            count += 1
            if query in line:
                matches.append((i, line))
    end = time()

    for i, line in matches:
        print(i, ': ', line, end='')
    print('Duration:', end - start)
    print(len(matches), 'of', count, 'lines match')
```

Python shell

```
...
3057752 : 195,,2,2,30,30,THR,HB,H,1,4.22,0.02,,1,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057753 : 196,,2,2,30,30,THR,HG21,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057754 : 197,,2,2,30,30,THR,HG22,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057755 : 198,,2,2,30,30,THR,HG23,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,228896,6203,2
Duration: 7.760039329528809
329 of 9758361 lines match
57557 :      "Well, then, to the West, or to England, or to Germany, where father
66515 :      kind master. He wanted me to go with his wife to Germany yesterday,
66642 :      of business in Germany in the past and my name is probably familiar
73273 :      associates with Germany. This he placed in his instrument cupboard.
Duration: 0.07700657844543457
4 of 76764 lines match
```

sudoku.py

```
class Sudoku:
    def __init__(self, puzzle):
        self.puzzle = puzzle

    def solve(self):
        def find_free():
            for i in range(9):
                for j in range(9):
                    if self.puzzle[i][j] == 0:
                        return (i, j)
            return None

        def unused(i, j):
            i_, j_ = i // 3 * 3, j // 3 * 3
            cells = {(i, k) for k in range(9)}
            cells |= {(k, j) for k in range(9)}
            cells |= {(i, j) for i in range(i_, i_ + 3)
                      for j in range(j_, j_ + 3)}
            return set(range(1, 10)) - {self.puzzle[i][j] for i, j in cells}

        class SolutionFound(Exception):
            pass

        def recursive_solve():
            cell = find_free()
            if not cell:
                raise SolutionFound
            i, j = cell
            for value in unused(i, j):
                self.puzzle[i][j] = value
                recursive_solve()
                self.puzzle[i][j] = 0

        try:
            recursive_solve()
        except SolutionFound:
            pass
```

sudoku.py (continued)

```
    def print(self):
        for i, row in enumerate(self.puzzle):
            cells = [f' {c} ' if c else ' . ' for c in row]
            print(''.join([''.join(cells[j:j+3]) for j in (0,3,6)]))
            if i in (2, 5):
                print('-----+-----+-----')

        with open('sudoku.txt') as f:
            A = Sudoku([[int(x) for x in line.strip()] for line in f])

        A.solve()
        A.print()
```

sudoku.txt

```
517600034
289004000
346205090
602000010
038006047
000000000
090000078
703400560
000000000
```

Python shell

```
| 5 1 7 | 6 9 8 | 2 3 4
| 2 8 9 | 1 3 4 | 7 5 6
| 3 4 6 | 2 7 5 | 8 9 1
|-----+-----+-----
| 6 7 2 | 8 4 9 | 3 1 5
| 1 3 8 | 5 2 6 | 9 4 7
| 9 5 4 | 7 1 3 | 6 8 2
|-----+-----+-----
| 4 9 5 | 3 6 2 | 1 7 8
| 7 2 3 | 4 8 1 | 5 6 9
| 8 6 1 | 9 5 7 | 4 2 3
```


match – case (since Python 3.10)

- Assume we want to do different things depending on the value of an expression (different *cases*)
- Can be done using `if`, but also using `match – case`, that is also evaluated top-down

match-case.py

```
x = 7
if x == 1:
    print('x is one')
elif x == 2:
    print('x is two')
elif x == 3 or x == 4 or x == 5:
    print('x is three, four or five')
else:
    print(x, 'is not in the range 1-5')
```

Python shell

```
| 7 is not in the range 1-5
```

match-case.py

```
x = 7
match x: # match expression
    case 1:
        print('x is one')
    case 2:
        print('x is two')
    case 3 | 4 | 5: # match any of the cases
        print('x is three, four or five')
    case value: # else, value = variable name
        print(value, 'is not in the range 1-5')
```

Python shell

```
| 7 is not in the range 1-5
```

match – case

Can match...

- simple values
- named variable values
- guards (`if`)
- sequences of values
- dictionaries
- builtin types
- user defined classes
- nested structures of the above

match-case.py

```
class Color:
    RED    = 'ff0000'
    GREEN  = '00ff00'
    BLUE   = '0000ff'

class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return f'Point({self.x}, {self.y})'
```

match-case.py

```
def f(x):
    match x:
        case 42:
            return 'the integer 42'
        case 1 | 2 | 3 | 4 | 5:
            return 'integer in range(1, 6)'
        case (1, 2):
            return 'sequence containing the elements 1 and 2'
        case [x, 2]:
            return 'sequence of length 2, last=2, first=' + str(x)
        case (x, y) if x + y == 7: # guard
            return 'sequence with two values with sum 7'
        case [0, 1, *x]: # x is list of remaining elements in sequence
            return 'sequence starting with 0 and 1, and tail ' + str(x)
        case {'a': 7, 'b': x}: # a dictionary containing at least two keys
            return 'dictionary "a" -> 7, "b" -> ' + str(x)
        case (('a' | 'b'), ('c' | 'd')):
            return 'tuple length 2, first "a" or "b", last "c" or "d"'
        case (('x' | 'y') as fst, ('x' | 'y') as snd):
            return '(fst, snd), where fst=' + str(fst) + ', snd=' + str(snd)
        case float(value): # test on builtin type
            return 'a float ' + str(value)
        case Color.RED: # class or object attribute
            return 'the color red'
        case Point(x=7, y=value): # Point object with attributes x and y
            return 'a Point object with x=7, and y=' + str(value)
        case Point(x, y): # requires __match_args__ in class Point
            return 'a point Point(' + str(x) + ', ' + str(y) + ')'
        case e: # ⚠ using the wildcard _ would not bind to a variable
            return 'cannot match ' + repr(e)
```

Python shell (match-case.py continued)

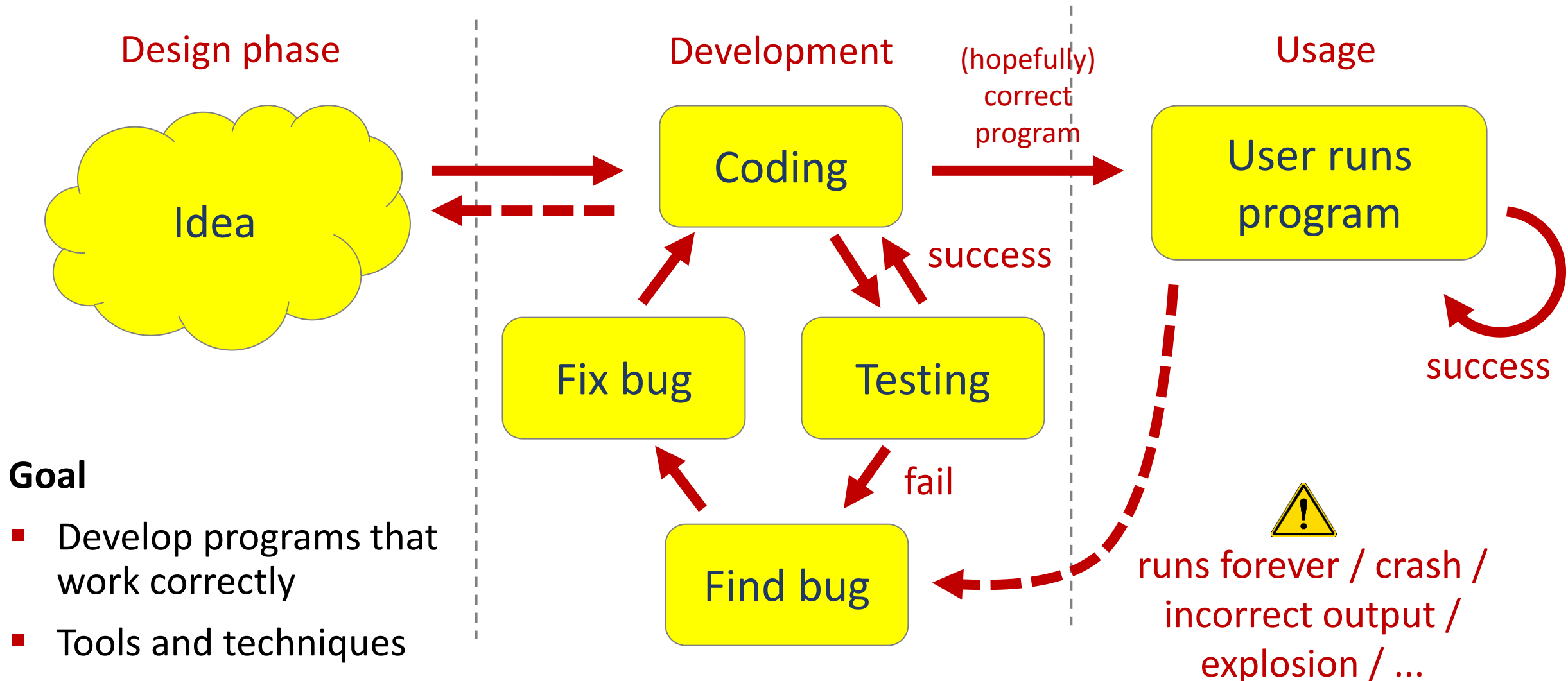
```
> for x in [42, 1, [1, 2], [7, 2], range(3, 5), (3, (5, 7)), (0, 1, 2, 3, 4, 5), {'a':7, 'b':42, 'c':1},  
           ('b', 'c'), ('y', 'x'), 3.14, 'ff0000', Point(7, 42), Point(3, 5), 'abc']:  
    print('f(' + repr(x) + ') = ' + repr(f(x)))  
f(42) = 'the integer 42'  
f(1) = 'integer in range(1, 6)'  
f([1, 2]) = 'sequence containing the elements 1 and 2'  
f([7, 2]) = 'sequence of length 2, last=2, first=7'  
f(range(3, 5)) = 'sequence with two values with sum 7'  
f((3, (5, 7))) = 'a triplet (3, (5, 7))'  
f((0, 1, 2, 3, 4, 5)) = 'sequence starting with 0 and 1, and tail [2, 3, 4, 5]'  
f({'a': 7, 'b': 42, 'c': 1}) = 'dictionary "a" -> 7, "b" -> 42'  
f(('b', 'c')) = 'tuple length 2, first "a" or "b", last "c" or "d" '  
f(('y', 'x')) = '(fst, snd), where fst=y, snd=x'  
f(3.14) = 'a float 3.14'  
f('ff0000') = 'the color red'  
f(Point(7, 42)) = 'a Point object with x=7, and y=42'  
f(Point(3, 5)) = 'a point Point(3, 5) '  
f('abc') = 'cannot match 'abc''
```

Documentation, testing and debugging

- docstring
- defensive programming
- assert
- test driven developement
- assertions
- testing
- unittest
- debugger
- coverage
- static type checking (mypy, pyright)

- On average, a developer creates 70 bugs per 1000 lines of code
- 15 bugs per 1,000 lines of code find their way to the customers
- Fixing a bug takes 30 times longer than writing a line of code
- 75% of a developer's time is spent on debugging

Ensuring good quality code ?



What is good code ?

- Readability
 - well-structured
 - documentation
 - comments
 - follow some standard structure (easy to recognize, follow [PEP8](#) Style Guide)
- Correctness
 - outputs the correct answer on valid input
 - eventually stops with an answer on valid input (should not go in infinite loop)
- Reusable...

Why ?

Documentation

- *specification of functionality*
- docstring
 - *for users of the code*
 - modules
 - methods
 - classes
- comments
 - *for readers of the code*

Testing

- Correct implementation ?
- Try to predict behavior on unknown input ?
- Performance guarantees ?

Debugging

- *Where is the `#!x$` bug ?*

Built-in exceptions (class hierarchy)

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError
    |   +-- IndexError
    |   +-- KeyError
    +-- MemoryError
    +-- NameError
    |   +-- UnboundLocalError
    +-- TypeError
    +-- ValueError
    |   +-- UnicodeError
    |       +-- UnicodeDecodeError
    |       +-- UnicodeEncodeError
    |       +-- UnicodeTranslateError
```

```
+-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |       +-- BrokenPipeError
    |       +-- ConnectionAbortedError
    |       +-- ConnectionRefusedError
    |       +-- ConnectionResetError
    +-- FileNotFoundError
    +-- InterruptedError
    +-- IsADirectoryError
    +-- NotADirectoryError
    +-- PermissionError
    +-- ProcessLookupError
    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
    |   +-- NotImplementedError
    |   +-- RecursionError
+-- SyntaxError
    |   +-- IndentationError
    |   +-- TabError
+-- SystemError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

Testing for unexpected behaviour ?

infinite-recursion1.py

```
def f(depth):  
    f(depth + 1)  # infinite recursion  
  
f(0)
```



Python shell

| `RecursionError: maximum recursion depth exceeded`

infinite-recursion2.py

```
def f(depth):  
    if depth > 100:  
        print('runaway recursion???)  
        raise SystemExit  # raise built-in exception  
    f(depth + 1)  
  
f(0)
```

Python shell

| runaway recursion???

infinite-recursion3.py

```
import sys  
  
def f(depth):  
    if depth > 100:  
        print('runaway recursion???)  
        sys.exit()  # system function  
    f(depth + 1)  
  
f(0)
```

raises SystemExit

Python shell

| runaway recursion???

- let the program eventually fail
- check and raise exceptions
- check and call `sys.exit`

Catching unexpected behaviour – `assert`

`infinite-recursion4.py`

```
def f(depth):  
    assert depth <= 100 # raise exception if False  
    f(depth + 1)  
  
f(0)
```

Python shell

```
| File "...\\infinite-recursion4.py", line 2, in f  
|     assert depth <= 100  
|     AssertionError
```

`infinite-recursion5.py`

```
def f(depth):  
    assert depth <= 100, 'runaway recursion???'  
    f(depth + 1)  
  
f(0)
```

Python shell

```
| File "...\\infinite-recursion5.py", line 2, in f  
|     assert depth <= 100, "runaway recursion???"  
|     AssertionError: runaway recursion???
```

- keyword **`assert`** checks if boolean expression is true, if not, raises exception **`AssertionError`**
- optional second parameter passed to the constructor of the exception
- try to fail fast to discover errors early – making debugging easier

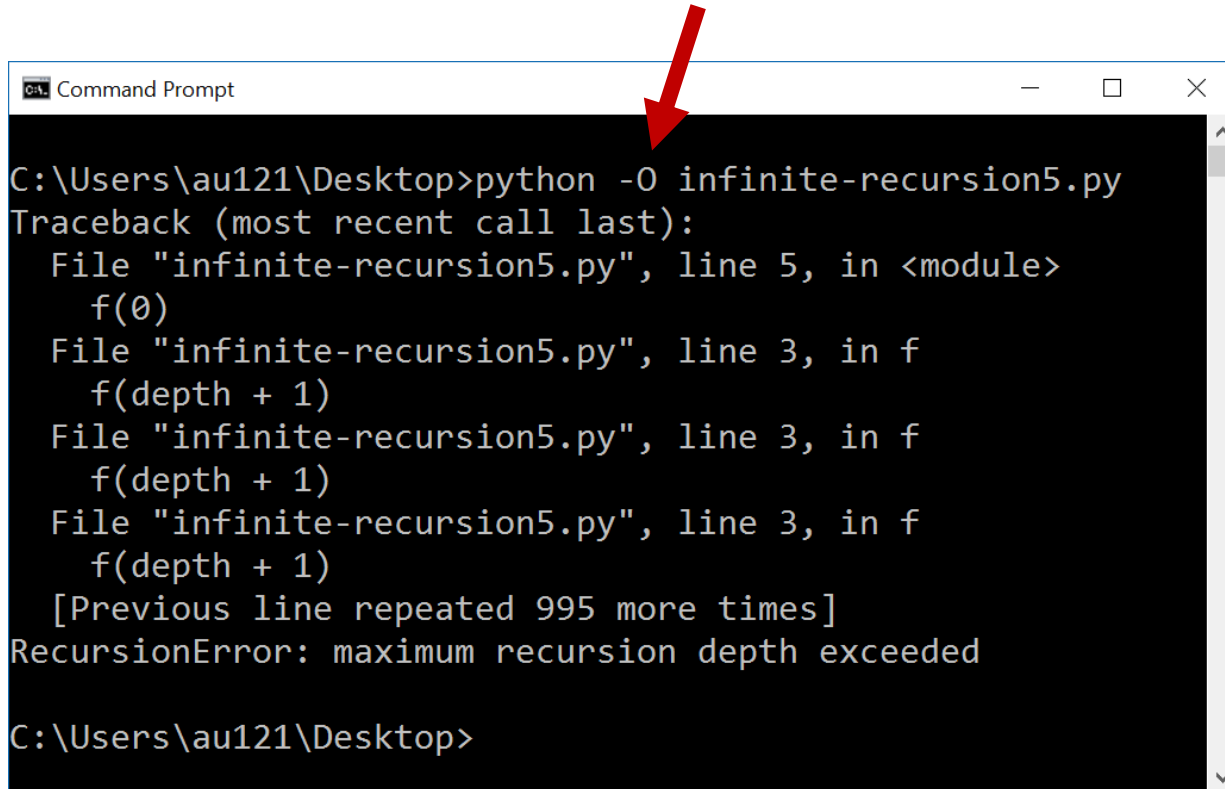
`infinite-recursion6.py`

```
def f(depth):  
    if not depth <= 100:  
        raise AssertionError('runaway recursion??')  
    f(depth + 1)  
  
f(0)
```

Python shell

```
| File "...\\infinite-recursion6.py", line 3, in f  
|     raise AssertionError("runaway recursion??")  
|     AssertionError: runaway recursion???
```

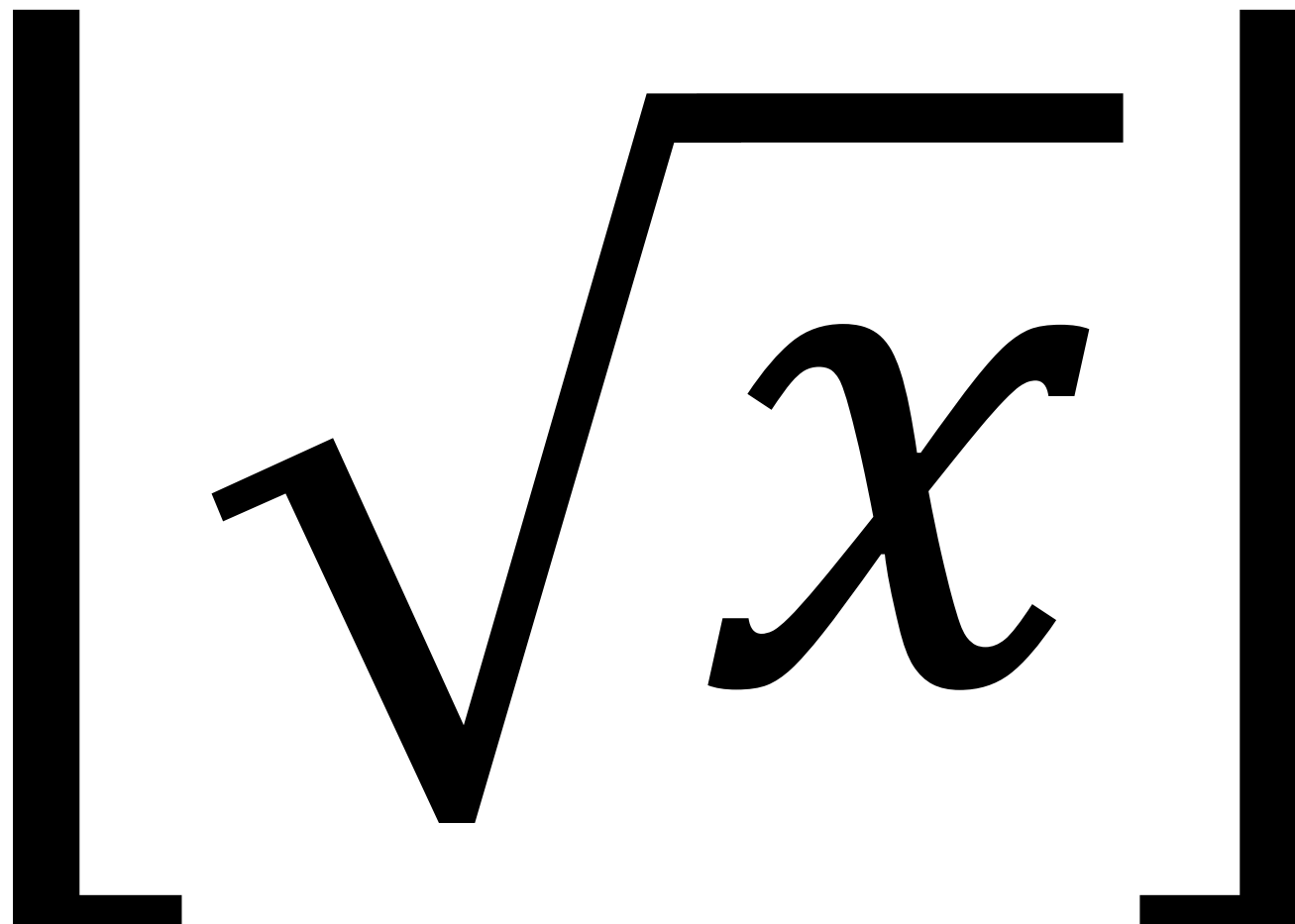
Disabling `assert` statements



```
Command Prompt
C:\Users\au121\Desktop>python -O infinite-recursion5.py
Traceback (most recent call last):
  File "infinite-recursion5.py", line 5, in <module>
    f(0)
  File "infinite-recursion5.py", line 3, in f
    f(depth + 1)
  File "infinite-recursion5.py", line 3, in f
    f(depth + 1)
  File "infinite-recursion5.py", line 3, in f
    f(depth + 1)
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
C:\Users\au121\Desktop>
```

- **`assert`** statements are good to help check correctness of program – but can **slow down** program
- invoking Python with option `-O` disables all assertions (by setting `__debug__` to `False`)

Example



First try... (seriously, the bugs were not on purpose)

intsqrt_buggy.py

```
def int_sqrt(x):  
    low = 0  
    high = x  
    while low < high - 1:  
        mid = (low + high) / 2  
        if mid ** 2 <= x:  
            low = mid  
        else:  
            high = mid  
    return low
```

Python shell

```
> int_sqrt(10)  
| 3.125 # 3.125 ** 2 = 9.765625  
> int_sqrt(-10)  
| 0 # what should the answer be ?
```

Let us add a specification...

intsqrt.py

```
def int_sqrt(x):  
    '''Compute the integer square root of an integer x.  
    Requires x >= 0 is an integer. ← input requirements  
    Returns the integer floor(sqrt(x)). ← output guarantees  
    ...  
'''
```

docstring

Python shell

```
> help(int_sqrt)  
| Help on function int_sqrt in module __main__:  
|  
| int_sqrt(x)  
|     Compute the integer square root of an integer x.  
|  
|     Requires x >= 0 is an integer.  
|     Returns the integer floor(sqrt(x)).
```

- all methods, classes, and modules can have a **docstring** (ideally have) as a **specification**
- for methods: summarize purpose in first line, followed by input requirements and output guarantees
- the docstring is assigned to the object's `__doc__` attribute

Let us check input requirements...

intsqrt.py

```
def int_sqrt(x):  
    '''Compute the integer square root of an integer x.  
  
    Requires x >= 0 is an integer.  
    Returns the integer floor(sqrt(x)).'''  
  
    assert isinstance(x, int) } check input  
    assert 0 <= x             } requirements  
    ...
```

Python shell

```
> int_sqrt(-10)  
| File "...\\int_sqrt.py", line 7, in int_sqrt  
|     assert 0 <= x  
| AssertionError
```

- doing explicit checks for valid input arguments is part of **defensive programming** and helps spotting errors early

(instead of continuing using likely wrong values... resulting in a final meaningless error)

Let us check if output correct...

intsqrt.py

```
def int_sqrt(x):  
    '''Compute the integer square root of an integer x.  
  
    Requires x >= 0 is an integer.  
    Returns the integer floor(sqrt(x)).'''  
  
    assert isinstance(x, int)  
    assert 0 <= x  
    ...  
    assert isinstance(result, int)  
    assert result ** 2 <= x < (result + 1) ** 2  
    return result
```

} check
output

Python shell

```
> int_sqrt(10)  
| File "...\\int_sqrt.py", line 20, in int_sqrt  
|     assert isinstance(result, int)  
| AssertionError
```

- output check identifies the error

`mid = (low + high) / 2`

- should have been

`mid = (low + high) // 2`

- The output check helps us to ensure that function specifications are satisfied in applications

Let us test some input values...

intsqrt.py

```
def int_sqrt(x):  
    ...  
  
assert int_sqrt(0) == 0  
assert int_sqrt(1) == 1  
assert int_sqrt(2) == 1  
assert int_sqrt(3) == 1  
assert int_sqrt(4) == 2  
assert int_sqrt(5) == 2  
assert int_sqrt(200) == 14
```

Python shell

```
| Traceback (most recent call last):  
|   File "...\\int_sqrt.py", line 28, in <module>  
|     assert int_sqrt(1) == 1  
|   File "...\\int_sqrt.py", line 21, in int_sqrt  
|     assert result ** 2 <= x < (result + 1) ** 2  
| AssertionError
```

- test identifies wrong output for x = 1

Let us check progress of algorithm...

intsqrt.py

```
...
low, high = 0, x
while low < high - 1: # low <= floor(sqrt(x)) < high
    assert low ** 2 <= x < high ** 2
    mid = (low + high) // 2
    if mid ** 2 <= x:
        low = mid
    else:
        high = mid
result = low
...
```

} check invariant
for loop
 $\lfloor \sqrt{x} \rfloor \in [low, high[$

Python shell

```
| Traceback (most recent call last):
|   File "... \int_sqrt.py", line 28, in <module>
|       assert int_sqrt(1) == 1
|   File "... \int_sqrt.py", line 21, in int_sqrt
|       assert result ** 2 <= x < (result + 1) ** 2
|   AssertionError
```

- test identifies
wrong output for $x = 1$
- but invariant apparently
correct ???
- problem
 $low == result == 0$
 $high == 1$
implies loop never entered
- output check identifies the
error
 $high = x$
- should have been
 $high = x + 1$

Final program

We have used **assertions** to:

- Test if **input** arguments / usage is valid (defensive programming)
- Test if computed **result** is correct
- Test if an internal **invariant** in the computation is satisfied
- Perform a **final test** for a set of test cases (should be run whenever we change anything in the implementation)

intsqrt.py

```
def int_sqrt(x):  
    '''Compute the integer square root of an integer x.  
  
    Requires x >= 0 is an integer.  
    Returns the integer floor(sqrt(x)).'''  
  
    assert isinstance(x, int)  
    assert 0 <= x  
  
    low, high = 0, x + 1  
    while low < high - 1: # low <= floor(sqrt(x)) < high  
        assert low ** 2 <= x < high ** 2  
        mid = (low + high) // 2  
        if mid ** 2 <= x:  
            low = mid  
        else:  
            high = mid  
    result = low  
  
    assert isinstance(result, int)  
    assert result ** 2 <= x < (result + 1) ** 2  
  
    return result  
  
assert int_sqrt(0) == 0  
assert int_sqrt(1) == 1  
assert int_sqrt(2) == 1  
assert int_sqrt(3) == 1  
assert int_sqrt(4) == 2  
assert int_sqrt(5) == 2  
assert int_sqrt(200) == 14
```

Which checks would you add to the below code?

binary-search.py

```
def binary_search(x, L):  
    '''Binary search for x in sorted list L.  
  
    Assumes x is an integer, and L a non-decreasing list of integers.  
  
    Returns index i,  $-1 \leq i < \text{len}(L)$ , where  $L[i] \leq x < L[i+1]$ ,  
    assuming  $L[-1] = -\text{infty}$  and  $L[\text{len}(L)] = +\text{infty}$ .'''  
  
    low, high = -1, len(L)  
    while low + 1 < high:  
        mid = (low + high) // 2  
        if x < L[mid]:  
            high = mid  
        else:  
            low = mid  
    result = low  
    return result
```

binary-search-assertions.py

```
def binary_search(x, L):
    '''Binary search for x in sorted list L.

    Assumes x is an integer, and L a non-decreasing list of integers.

    Returns index i, -1 <= i < len(L), where L[i] <= x < L[i+1],
    assuming L[-1] = -infty and L[len(L)] = +infty.'''

    input {
        assert isinstance(x, int)
        assert isinstance(L, list)
        assert all([isinstance(e, int) for e in L])
        assert all([L[i] <= L[i + 1] for i in range(len(L) - 1)])
    } ① inefficient ⚠

    low, high = -1, len(L)
    while low + 1 < high: # L[low] <= x < L[high]
        input loop {
            assert (low == -1 or L[low] <= x) and (high == len(L) or x < L[high])
            mid = (low + high) // 2
            {
                assert isinstance(L[mid], int)
                assert (low == -1 or L[low] <= L[mid]) and (high == len(L) or L[mid] <= L[high])
            } ②
            if x < L[mid]:
                high = mid
            else:
                low = mid
        }
    result = low

    output {
        assert (isinstance(result, int) and -1 <= result < len(L) and
            ((result == -1 and (len(L) == 0 or x < L[0])) or
             (result == len(L) - 1 and x >= L[-1]) or
             (0 <= result < len(L) - 1 and L[result] <= x < L[result + 1])))
    }

    return result

assert binary_search(42, []) == -1
assert binary_search(42, [7]) == 0
assert binary_search(7, [42]) == -1
assert binary_search(7, [42, 42, 42]) == -1
assert binary_search(42, [7, 7, 7]) == 2
assert binary_search(42, [7, 7, 7, 56, 81]) == 2
assert binary_search(8, [1, 3, 5, 7, 9]) == 3
} test cases
```

- ① Verifying if L is a sorted list of integers can slow down the program significantly
- ② Alternative is to only verify if the part of L visited is a sorted subsequence

Test driven development / Stress tests / Random testing

■ Test driven development

Write the tests before functionality
– only write code needed by tests

■ The challenge – what tests to do?

Can you manually find all relevant cases? In particular all edge cases?

■ Automate the testing?

- Write method that can verify the output (possibly slower than the method)
- Systematically try *all* possible inputs (if range is small)
- Try a large random subset of inputs (if many possible inputs)

intsqrt_automatic_testing.py

```
import random

def int_sqrt(x):
    return 42 # Dummy code - write test code first

def test_int_sqrt(x):
    print('.', end='', flush=True) # Show progress
    assert x >= 0 # Verify input
    answer = int_sqrt(x)
    # Verify output
    assert answer ** 2 <= x < (answer + 1) ** 2

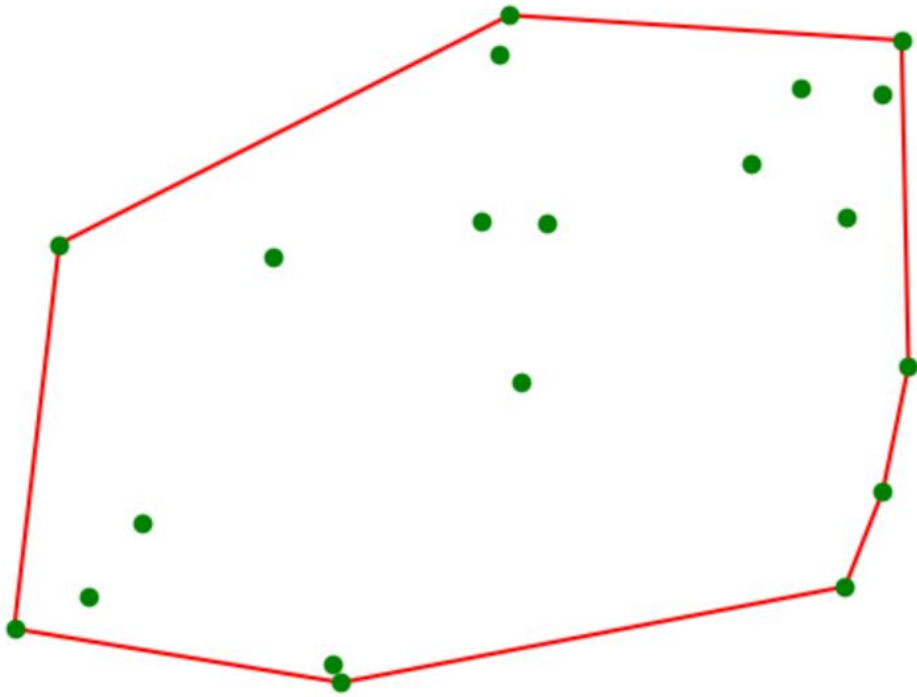
# Test small inputs
for x in range(0, 100):
    test_int_sqrt(x)

# Test increasing sized inputs
for d in range(3, 30):
    for _ in range(100): # Repeat for each size
        test_int_sqrt(random.randint(1, 10 ** d))
```

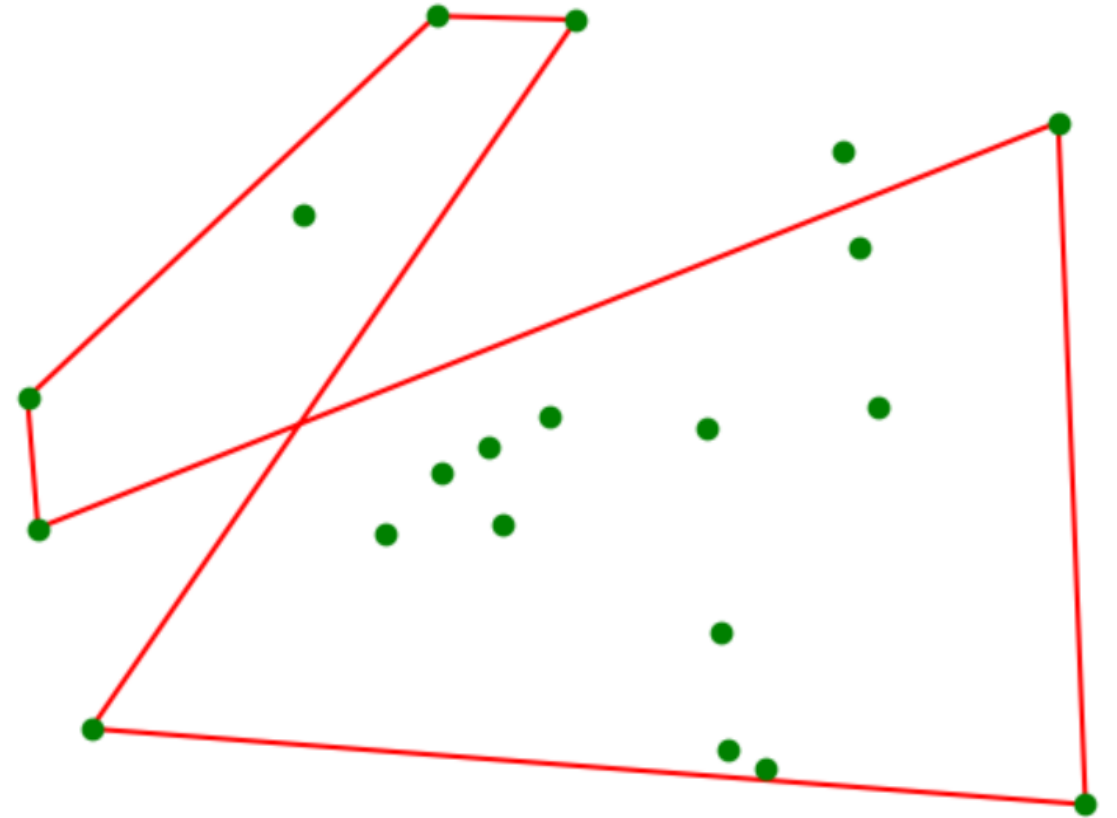
Testing – how ?

- Run set of test cases
 - test all cases in input/output specification (**black box testing**)
 - test all special cases (**black box testing**)
 - set of tests should force all lines of code to be tested (**glass box testing**)
- Visual test
- Automatic testing
 - Systematically / randomly generate input instances
 - Create function to **validate** if output is correct (hopefully easier than finding the solution)
- Formal verification
 - Use computer programs to do formal proofs of correctness, like using Coq

Visual testing – Convex hull computation



Correct



**Bug !
(not convex)**

doctest

- Python module
- Test instances (pairs of input and corresponding output) are written in the doc strings, formatted as in an interactive Python session

binary-search-doctest.py

```
def binary_search(x, L):  
    '''Binary search for x in sorted list L.  
  
    Examples:  
    >>> binary_search(42, [])  
    -1  
    >>> binary_search(42, [7])  
    0  
    >>> binary_search(42, [7,7,7,56,81])  
    2  
    >>> binary_search(8, [1,3,5,7,9])  
    3  
    '''  
  
    low, high = -1, len(L)  
    while low + 1 < high:  
        mid = (low + high) // 2  
        if x < L[mid]:  
            high = mid  
        else:  
            low = mid  
    return low  
  
import doctest  
doctest.testmod(verbose=True)
```

Python shell

```
Trying:  
    binary_search(42, [])  
Expecting:  
    -1  
ok  
Trying:  
    binary_search(42, [7])  
Expecting:  
    0  
ok  
Trying:  
    binary_search(42, [7,7,7,56,81])  
Expecting:  
    2  
ok  
Trying:  
    binary_search(8, [1,3,5,7,9])  
Expecting:  
    3  
ok  
1 items had no tests:  
    __main__  
1 items passed all tests:  
    4 tests in __main__.binary_search  
4 tests in 2 items.  
4 passed and 0 failed.  
Test passed.
```

pytest

- Run all tests stored in functions prefixed by `test_` or `test_` prefixed test methods inside `Test` prefixed test classes
- `pip install pytest`
- Run the `pytest` program from a shell

pytest.org

binary-search-pytest.py

```
import pytest

def binary_search(x, L):
    '''Binary search for x in sorted list L.'''
    low, high = -1, len(L)
    while low + 1 < high:
        mid = (low + high) // 2
        if x < L[mid]:
            high = mid
        else:
            low = mid
    return low

def test_binary_search():
    assert binary_search(42, []) == -1
    assert binary_search(42, [7]) == 0
    assert binary_search(42, [7,7,7,56,81]) == 2
    assert binary_search(8, [1,3,5,7,9]) == 3

def test_types():
    with pytest.raises(TypeError):
        _ = binary_search(5, ['a', 'b', 'c'])
```

Shell

```
> pytest binary-search-pytest.py
| ===== test session starts =====
| platform win32 -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
| plugins: anyio-3.6.2
| collected 2 items
|
| binary-search-pytest.py .. [100%]
| ===== 2 passed in 0.05s =====
```

unittest

- Python module
- A comprehensive **object-oriented test framework**, inspired by the corresponding JUnit test framework for Java

binary-search-unittest.py

```
def binary_search(x, L):
    '''Binary search for x in sorted list L.'''

    low, high = -1, len(L)
    while low + 1 < high:
        mid = (low + high) // 2
        if x < L[mid]:
            high = mid
        else:
            low = mid
    return low

import unittest

class TestBinarySearch(unittest.TestCase):
    def test_search(self):
        self.assertEqual(binary_search(42, []), -1)
        self.assertEqual(binary_search(42, [7]), 0)
        self.assertEqual(binary_search(42, [7,7,7,56,81]), 2)
        self.assertEqual(binary_search(8, [1,3,5,7,9]), 3)

    def test_types(self):
        self.assertRaises(TypeError, binary_search, 5, ['a', 'b', 'c'])

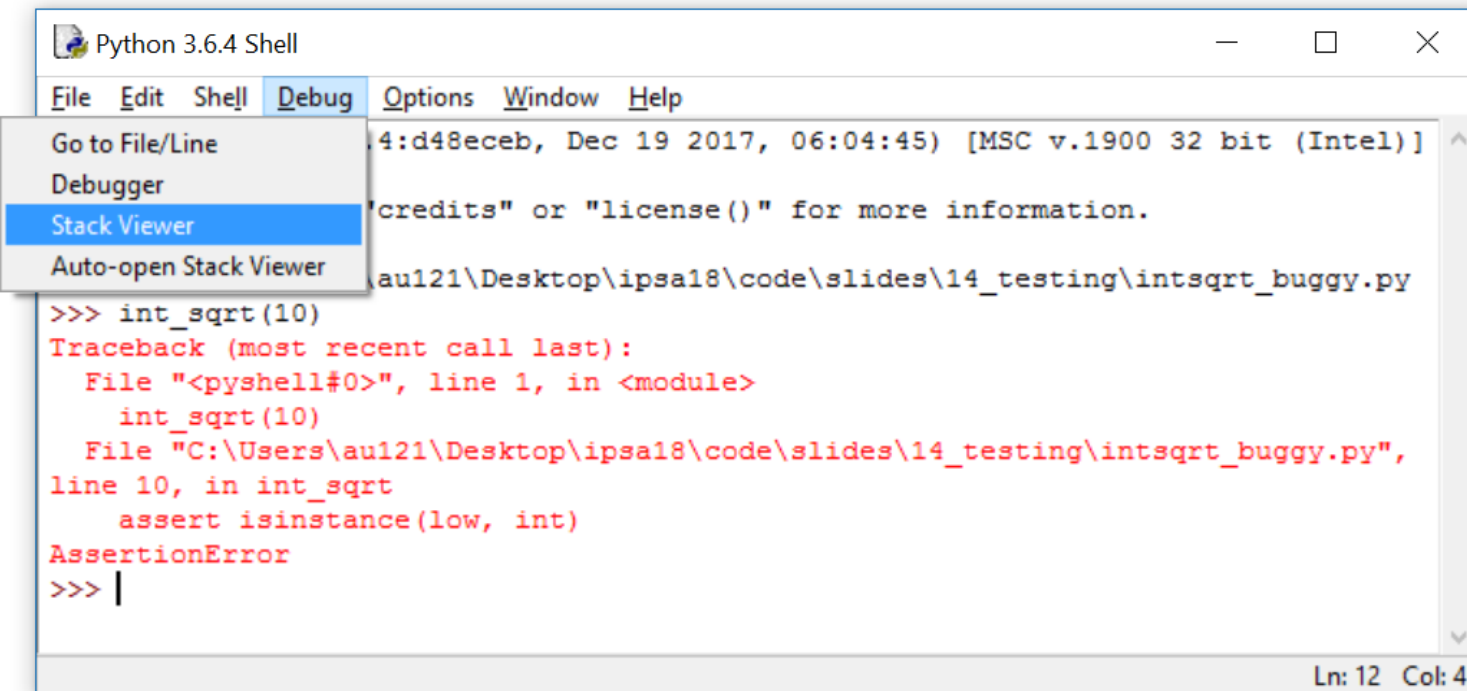
unittest.main(verbosity=2)
```

Python shell

```
| test_search (__main__.TestBinarySearch) ... ok
| test_types (__main__.TestBinarySearch) ... ok
| -----
| Ran 2 tests in 0.051s
| OK
```

Debugger (IDLE)

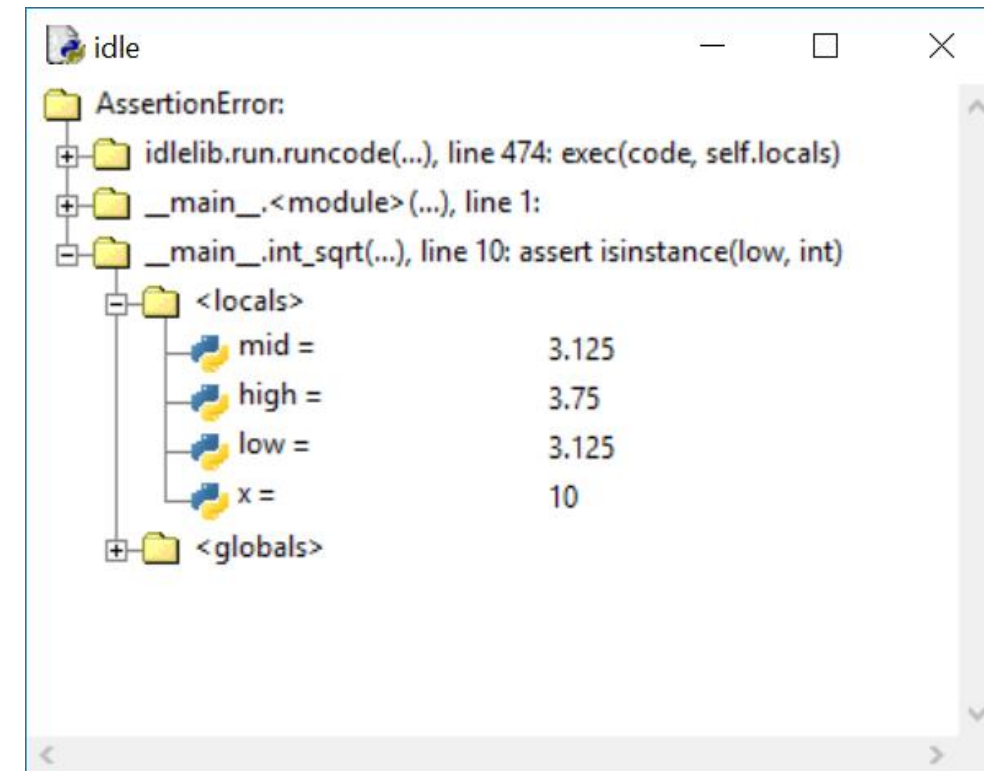
- When an exception has stopped the program, you can examine the state of the variables using **Debug > Stack Viewer** in the Python shell



The screenshot shows the Python 3.6.4 Shell window. The 'Debug' menu is open, with 'Stack Viewer' highlighted. The shell displays a traceback for an `AssertionError` that occurred in `int_sqrt(10)`. The traceback shows the call stack starting from the shell, through the module, and into the `int_sqrt` function at line 10, where the assertion `assert isinstance(low, int)` failed.

```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Go to File/Line
Debugger
Stack Viewer
Auto-open Stack Viewer
4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
credits" or "license()" for more information.
au121\Desktop\ipsa18\code\slides\14_testing\intsqrt_buggy.py
>>> int_sqrt(10)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    int_sqrt(10)
  File "C:\Users\au121\Desktop\ipsa18\code\slides\14_testing\intsqrt_buggy.py",
line 10, in int_sqrt
    assert isinstance(low, int)
AssertionError
>>> |
```

Ln: 12 Col: 4



The screenshot shows the Stack Viewer window in IDLE. It displays the call stack for the current exception. The stack frames are:

- `idlelib.run.runcode(...), line 474: exec(code, self.locals)`
- `_main_.<module>(...), line 1:`
- `_main_.int_sqrt(...), line 10: assert isinstance(low, int)`

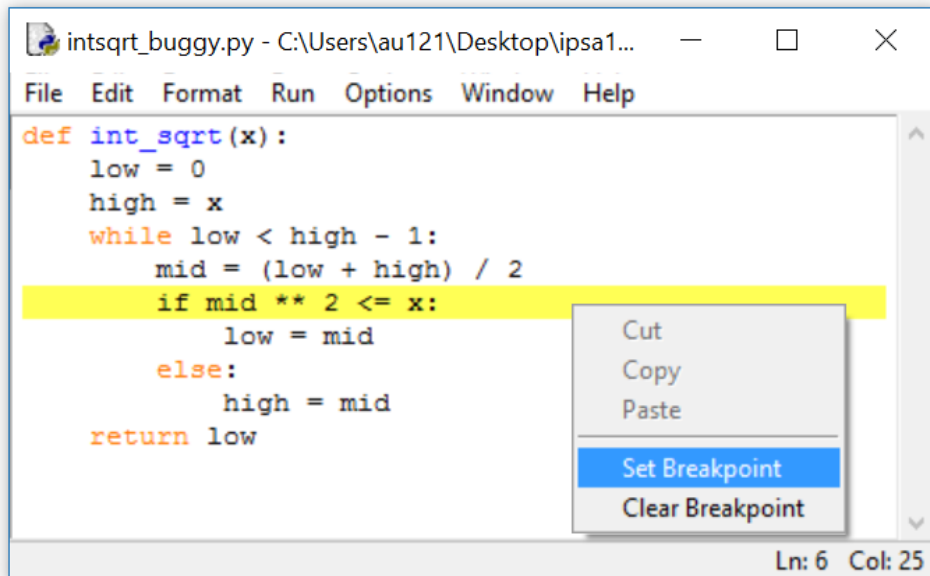
The current frame is `_main_.int_sqrt(...)`. It shows the local variables:

Variable	Value
<code>mid</code>	<code>3.125</code>
<code>high</code>	<code>3.75</code>
<code>low</code>	<code>3.125</code>
<code>x</code>	<code>10</code>

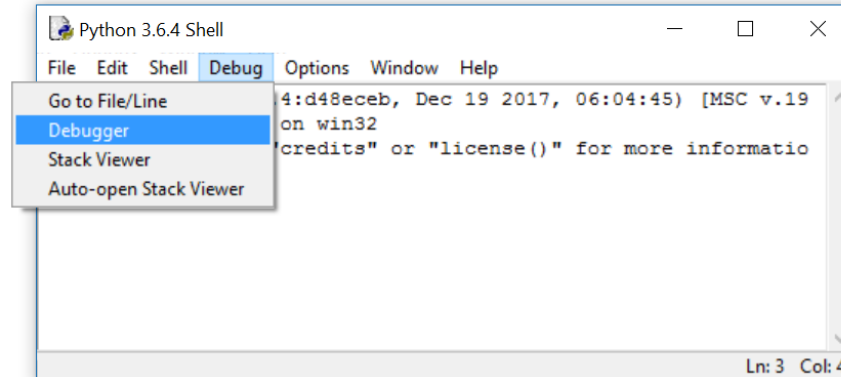
The global variables section is also visible, showing `<globals>`.

Stepping through a program (IDLE debugger)

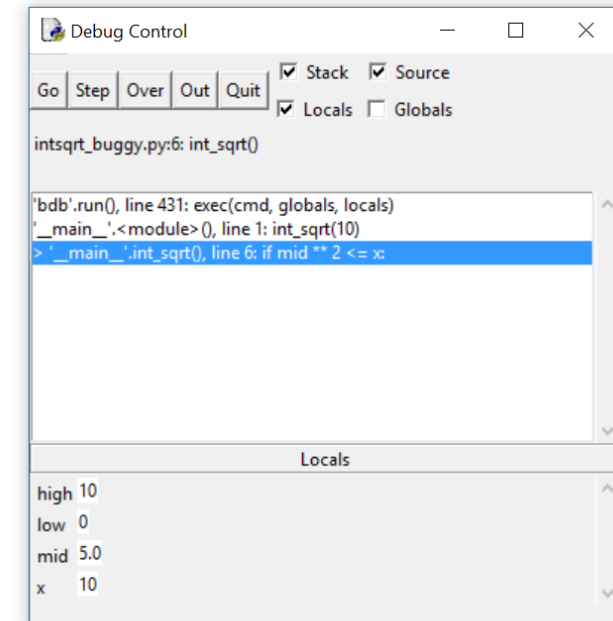
- **Debug > Debugger** in the Python shell opens Debug Control window
- **Right click** on a code line in editor to set a “breakpoint” in your code
- **Debug Control:** Go → run until next breakpoint is encountered;
Step → execute one line of code; Over → run function call without details;
Out → finish current function call; Quit → Stop program;



```
def int_sqrt(x):  
    low = 0  
    high = x  
    while low < high - 1:  
        mid = (low + high) / 2  
        if mid ** 2 <= x:  
            low = mid  
        else:  
            high = mid  
    return low
```



```
Python 3.6.4 Shell  
File Edit Shell Debug Options Window Help  
4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.19  
on win32  
credits" or "license()" for more informatio
```



```
Debug Control  
Go Step Over Out Quit [x] Stack [x] Source  
[x] Locals [ ] Globals  
intsqrt_buggy.py:6: int_sqrt()  
'bdb'.run(), line 431: exec(cmd, globals, locals)  
'._main_'.<module>(), line 1: int_sqrt(10)  
> '._main_'.int_sqrt(), line 6: if mid ** 2 <= x:  
  
Locals  
high 10  
low 0  
mid 5.0  
x 10
```

Coverage

- Ensure that your tests cover the whole code and all possible branches are taken
- The module `coverage` can monitor running your code and report which lines and branches were not executed
- `pip install coverage`
- **Note** 100% coverage does not guarantee that there are no errors... just fewer

goldbach.py

```
1 def odd(x):
2     return x % 2 == 1

3 def sum_of_three_primes(n):
4     assert odd(n) and n > 5
5     primes = (set(range(2, n + 1)) -
6               set(x for f in range(2, n + 1)
7                   for x in range(2 * f, n + 1, f)))
8     for x in primes:
9         for y in primes:
10            for z in primes:
11                if n == x + y + z:
12                    print(n, 'is the sum of three primes', x, y, z)
13            return
14    print(n, 'is not the sum of three primes')

15 for n in range(7, 1000, 2):
16     sum_of_three_primes(n)
```

Shell

```
> coverage run --branch goldbach.py
7 is the sum of three primes 2 2 3
9 is the sum of three primes 2 2 5
...
999 is the sum of three primes 3 5 991
> coverage report -m goldbach.py
```

Name	Stmts	Miss	Branch	BrPart	Cover	Missing
goldbach.py	14	1	12	1	92%	14
TOTAL	14	1	12	1	92%	

pypi.org/project/coverage

en.wikipedia.org/wiki/Goldbach's_weak_conjecture

coverage html

Coverage for **goldbach.py**: 92%

14 statements

13 run

1 missing

0 excluded

1 partial

« prev

^ index

» next

coverage.py v6.5.0, created at 2022-10-05 17:23 +0200

```
1 def odd(x):
2     return x % 2 == 1
3 def sum_of_three_primes(n):
4     assert odd(n) and n > 5
5     primes = (set(range(2, n + 1)) -
6               set(x for f in range(2, n + 1)
7                   for x in range(2 * f, n + 1, f)))
8     for x in primes:
9         for y in primes:
10            for z in primes:
11                if n == x + y + z:
12                    print(n, 'is the sum of three primes', x, y, z)
13                return
14     print(n, 'is not the sum of three primes')
15 for n in range(7, 1001, 2):
16     sum_of_three_primes(n)
```

8 ↔ 14

line 8 didn't jump to line 14, because the loop on line 8 didn't complete

Concluding remarks

- Simple debugging: add print statements
- **Test driven development** → Strategy for code development, where tests are written before the code
- **Defensive programming** → add tests (assertions) to check if input/arguments are valid according to specification
- When designing tests, ensure **coverage**
(the set of test cases should make sure all code lines get executed)
- **Python testing frameworks: doctest, unittest, pytest, ...**

Mypy – a static type checker for Python

Experimental 

- **Static type checking** tries to analyze a program for potential type errors **without** executing the program
- Installing:

```
pip install mypy
```
- Running Python will cause an error during execution, whereas using **mypy** the error will be found without executing the program
- Standard (and required) in statically typed languages like Java, C, C++

mypy-simple.py

```
print('start')
print(42 + 'abc') # error
print('end')
```

Shell

```
> python mypy-simple.py
| start
| TypeError: unsupported operand type(s)
| for +: 'int' and 'str'
> mypy mypy-simple.py
| mypy-simple.py:2: error: Unsupported
| operand types for + ("int" and "str")
| [operator]
```

mypy does not spot all errors...

mypy-add.py

```
def add(x, y):  
    return x + y # bug: x int and y string  
print(add(42, 'abc'))
```

Shell

```
> python add.py  
| TypeError: unsupported operand type(s) for +: 'int' and 'str'  
> mypy add.py  
| Success: no issues found in 1 source file 
```

Type hints (PEP 484)

- Python allows type hints in programs
- Type hints are **ignored** at run-time by Python, but useful for static type analysis (e.g. mypy)
- Syntax

variable : *type*

variable : *type* = value

mypy-basic-types.py

```
x : int # type hint
x = 42
x = 'abc' # type error
y : int = 42 # type hint
y = 'abc' # type error
z = 42
z = 'abc' # type changed from int to str
print(x, y, z)
```

Shell

```
> python mypy-basic-types.py
| abc abc abc
> mypy mypy-basic-types.py
| mypy-basic-types.py:3: error: Incompatible
| types in assignment (expression has type
| "str", variable has type "int")
| mypy-basic-types.py:5: error: ...
| mypy-basic-types.py:7: error: ...
```

Type hints – functions

`def name(variable : type, ...) -> return type:`

mypy-function.py	Shell
<pre>def f(x: int, units: str) -> str: return str(x) + ' ' + units def g(x, units: str) -> str: return str(x) + ' ' + units print(f(3, 'cm')) print(f('one', 'meter')) print(g(3, 'cm')) print(g('one', 'meter')) print(f.__annotations__)</pre>	<pre>> python mypy-function.py 3 cm one meter 3 cm one meter {'x': <class 'int'>, 'units': <class 'str'>, 'return': <class 'str'>} > mypy mypy-function.py mypy-function.py:8: error: Argument 1 to "f" has incompatible type "str"; expected "int"</pre>

- For functions and methods `function.__annotations__` is a dictionary with the annotation
- The types become part of the documentation

More type hints in Python 3.9

...see PEP 484 for even more...

mypy-typing.py

```
from typing import Mapping, Set, List, Tuple, Union, Optional

S : Set = {}                                # error {} dictionary
S2 : Set[int] = {1, 2, 'abc'}               # error 'abc' is not int
D : Mapping[int, int] = {1: 42, 'a': 1}     # error 'a' is not int
T : Tuple[int, str] = (42, 7)               # error 7 is not str
L : List[Union[int, str]] = [42, 'a', None] # error list can only contain int and str
L2 : List[Optional[str]] = ['abc', None, 42] # error list can only contain str og None
```

Shell

```
> mypy mypy-typing.py
| mypy-typing.py:3: error: Incompatible types in assignment (expression has type "Dict[<nothing>, <nothing>]", variable has type "Set[Any]")
| mypy-typing.py:4: error: Argument 3 to <set> has incompatible type "str"; expected "int"
| mypy-typing.py:5: error: Dict entry 1 has incompatible type "str": "int"; expected "int": "int"
| mypy-typing.py:6: error: Incompatible types in assignment (expression has type "Tuple[int, int]", variable has type "Tuple[int, str]")
| mypy-typing.py:7: error: List item 2 has incompatible type "None"; expected "Union[int, str]"
| mypy-typing.py:8: error: List item 2 has incompatible type "int"; expected "Optional[str]"
```

... the same in Python 3.10

mypy-typing-new.py

```
# deprecated: from typing import Mapping, Set, List, Tuple, Union, Optional

S : set = {}                                # error {} dictionary
S2 : set[int] = {1, 2, 'abc'}               # error 'abc' is not int
D : dict[int, int] = {1: 42, 'a': 1}         # error 'a' is not int
T : tuple[int, str] = (42, 7)               # error 7 is not str
L : list[int | str] = [42, 'a', None]        # error list can only contain int and str
L2 : list[str | None] = ['abc', None, 42]    # error list can only contain str og None
```

Shell

```
> mypy mypy-typing-new.py
| mypy-typing-new.py:3: error: Incompatible types in assignment (expression has type "Dict[<nothing>, <nothing>]", variable has type "Set[Any]")
| mypy-typing-new.py:4: error: Argument 3 to <set> has incompatible type "str"; expected "int"
| mypy-typing-new.py:5: error: Dict entry 1 has incompatible type "str": "int"; expected "int": "int"
| mypy-typing-new.py:6: error: Incompatible types in assignment (expression has type "Tuple[int, int]", variable has type "Tuple[int, str]")
| mypy-typing-new.py:7: error: List item 2 has incompatible type "None"; expected "Union[int, str]"
| mypy-typing-new.py:8: error: List item 2 has incompatible type "int"; expected "Optional[str]"
```

Specific values

mypy-literal.py

```
from typing import Literal

def calc(cmd: Literal['add', 'sub'], x: int, y: int) -> int:
    match cmd:
        case 'add': return x + y
        case 'sub': return x - y
        case _: raise ValueError(f"Unknown command '{cmd}'")

print(f"{calc('add', 5, 8) = }")
print(f"{calc('sub', 5, 8) = }")
print(f"{calc('mul', 5, 8) = }") # error
```

Shell

```
> python.exe mypy-literal.py
| calc('add', 5, 8) = 13
| calc('sub', 5, 8) = -3
| ValueError: Unknown command 'mul'
> mypy.exe .\mypy-literal.py
| mypy-literal.py:11: error: Argument 1 to "calc" has incompatible type "Literal['mul']";
| expected "Literal['add', 'sub']" [arg-type]
| Found 1 error in 1 file (checked 1 source file)
```


Type hints for methods with multiple signatures

printing1.py

```
class MyClass:
    def print(self, value: int | str) -> None:
        if isinstance(value, int):
            print('An integer', value)
        elif isinstance(value, str):
            print('A string', value)
```

- The right solution is useful for functions/methods with more complex overloaded type signatures
- `...` is the Python Ellipsis object
- `@overload` is a Python *decorator*

printing2.py

```
from typing import overload

class MyClass:
    # type definition of usages
    @overload
    def print(self, value: int) -> None: ...

    @overload
    def print(self, value: str) -> None: ...

    # actual implementation
    def print(self, value):
        if isinstance(value, int):
            print('An integer', value)
        elif isinstance(value, str):
            print('A string', value)
```

Type hints for class inheritance

```
abstract.py

from typing import override, final

class A():
    @final # prevent subclasses to override f (since Python 3.8)
    def f(self):
        print('f')
        self.g()

    def g(self):
        raise NotImplementedError

class B(A):
    @override # check if parent class contains g (since Python 3.12)
    def g(self):
        print('g')
```

- Use **pyright** (`pip install pyright`) to check the above, a static type checking tool from Microsoft
- mypy does not check it (mypy 1.8.0)

Decorators

- @

Course overview

Basic programming
Advanced / specific python
Libraries & applications

1. Introduction to Python	10. Functions as objects	19. Linear programming
2. Python basics / if	11. Object oriented programming	20. Generators, iterators, with
3. Basic operations	12. Class hierarchies	21. Modules and packages
4. Lists / while / for	13. Exceptions and files	22. Working with text
5. Tuples / comprehensions	14. Doc, testing, debugging	23. Relational data
6. Dictionaries and sets	15. Decorators	24. Clustering
7. Functions	16. Dynamic programming	25. Graphical user interfaces (GUI)
8. Recursion	17. Visualization and optimization	26. Java vs Python
9. Recursion and Iteration	18. Multi-dimensional data	27. Final lecture

10 handins
1 final project (last 1 month)

Python decorators are just syntactic sugar

Python

```
@dec2
@dec1
def func(arg1, arg2, ...):
    pass
```

≡

Python

```
def func(arg1, arg2, ...):
    pass

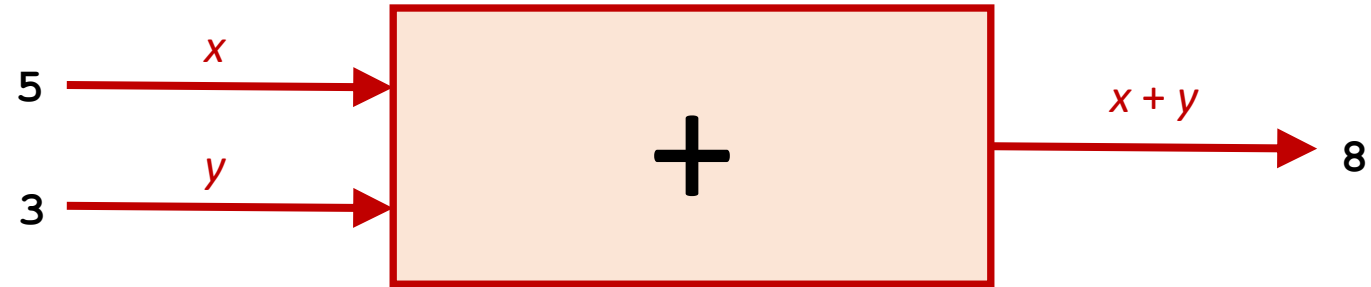
func = dec2(dec1(func))
```

'pie-decorator' syntax

dec1, dec2, ... are functions (decorators) taking a *function as an argument* and *returning a new function*

Note: decorators are listed bottom up in order of execution

Recap functions



Contrived example : Plus one (I-II)

plus_one1.py

```
def plus_one(x):  
    return x + 1  
  
def square(x):  
    return x ** 2  
  
def cube(x):  
    return x ** 3  
  
print(plus_one(square(5)))  
print(plus_one(cube(5)))
```

Python shell

```
| 26  
| 126
```

Assume we *always* need to call `plus_one` on the result of `square` and `cube` (don't ask why!)

plus_one2.py

```
def plus_one(x):  
    return x + 1  
  
def square(x):  
    return plus_one(x ** 2)  
  
def cube(x):  
    return plus_one(x ** 3)  
  
print(square(5))  
print(cube(5))
```

Python shell

```
| 26  
| 126
```

We could call `plus_one` inside functions (but could be more `return` statements in functions)

Contrived example : Plus one (III-IV)

plus_one3.py

```
def plus_one(x):  
    return x + 1  
  
def square(x):  
    return x ** 2  
  
def cube(x):  
    return x ** 3  
  
square_original = square  
cube_original = cube  
  
square = lambda x: plus_one(square_original(x))  
cube = lambda x: plus_one(cube_original(x))  
  
print(square(5))  
print(cube(5))
```

Python shell

```
| 26  
| 126
```

Overwrite square and cube with decorated versions

plus_one4.py

```
def plus_one(x):  
    return x + 1  
  
def plus_one_decorator(f):  
    return lambda x: plus_one(f(x))  
  
def square(x):  
    return x ** 2  
  
def cube(x):  
    return x ** 3  
  
square = plus_one_decorator(square)  
cube = plus_one_decorator(cube)  
  
print(square(5))  
print(cube(5))
```

Python shell

```
| 26  
| 126
```

Create a decorator function `plus_one_decorator`

Contrived example : Plus one (V-VI)

plus_one5.py

```
def plus_one(x):  
    return x + 1  
  
def plus_one_decorator(f):  
    return lambda x: plus_one(f(x))  
  
@plus_one_decorator  
def square(x):  
    return x ** 2  
  
@plus_one_decorator  
def cube(x):  
    return x ** 3  
  
print(square(5))  
print(cube(5))
```

Python shell

```
| 26  
| 126
```

Use Python **decorator syntax**

plus_one6.py

```
def plus_one_decorator(f):  
    def plus_one(x):  
        return f(x) + 1  
  
    return plus_one  
  
@plus_one_decorator  
def square(x):  
    return x ** 2  
  
@plus_one_decorator  
def cube(x):  
    return x ** 3  
  
print(square(5))  
print(cube(5))
```

Python shell

```
| 26  
| 126
```

Create **local function** instead of using `lambda`

Contrived example : Plus one (VII)

plus_one7.py

```
def plus_one_decorator(f):  
    def plus_one(x):  
        return f(x) + 1  
    return plus_one  
  
@plus_one_decorator  
@plus_one_decorator  
def square(x):  
    return x ** 2  
  
@plus_one_decorator  
@plus_one_decorator  
@plus_one_decorator  
def cube(x):  
    return x ** 3  
  
print(square(5))  
print(cube(5))
```

Python shell

```
| 27  
| 128
```

- A function can have an arbitrary number of decorators (also the same repeated)
- Decorators are listed bottom up in order of execution

Handling arguments

run_twice1.py

```
def run_twice(f):  
    def wrapper():  
        f()  
        f()  
    return wrapper  
  
@run_twice  
def hello_world():  
    print('Hello world')  
  
hello_world()
```

Python shell

```
| Hello world  
| Hello world
```

“wrapper” is a common name for the function returned by a decorator

run_twice2.py

```
def run_twice(f):  
    def wrapper(*args):  
        f(*args)  
        f(*args)  
    return wrapper  
  
@run_twice  
def hello_world():  
    print('Hello world')  
  
@run_twice  
def hello(txt):  
    print('Hello', txt)  
  
hello_world()  
hello('Mars')
```

Python shell

```
| Hello world  
| Hello world  
| Hello Mars  
| Hello Mars
```

args holds the arguments in a tuple given to the function to be decorated

Question – What does the decorated program print ?

decorator_quiz.py

```
def double(f):  
    def wrapper(*args):  
        return 2 * f(*args)  
    return wrapper  
  
def add_three(f):  
    def wrapper(*args):  
        return 3 + f(*args)  
    return wrapper  
  
@double  
@add_three  
def seven():  
    return 7  
  
print(seven())
```

- 7
- 10
- 14
- 17
- 20
- Don't know



Example: Enforcing argument types

- Defining decorators can be (slightly) complicated
- Using decorators is easy

`integer_sum1.py`

```
def integer_sum(*args):  
    assert all([isinstance(x, int) for x in args]),\  
            'all arguments must be int'  
    return sum(args)
```

`Python shell`

```
> integer_sum(1, 2, 3, 4)  
| 10  
> integer_sum(1, 2, 3.2, 4)  
| AssertionError: all arguments must be int
```

`integer_sum2.py`

```
def enforce_integer(f):    # decorator function  
    def wrapper(*args):  
        assert all([isinstance(x, int) for x in args]),\  
                'all arguments must be int'  
        return f(*args)  
    return wrapper  
  
@enforce_integer  
def integer_sum(*args):  
    return sum(args)
```

`Python shell`

```
> integer_sum(1, 2, 3, 4)  
| 10  
> integer_sum(1, 2, 3.2, 4)  
| AssertionError: all arguments must be int
```

Decorators can take arguments

Python

```
@dec(argA, argB, ...)  
def func(arg1, arg2, ...):  
    pass
```

=

Python

```
def func(arg1, arg2, ...):  
    pass  
func = dec(argA, argB, ...)(func)
```

dec is a function (decorator) that takes a *list of arguments* and *returns a function* (to decorate `func`) that takes a *function as an argument* and *returns a new function*

Example: Generic type enforcing

print_repeated.py

```
def enforce_types(*decorator_args):
    def decorator(f):
        def wrapper(*args):
            assert len(args) == len(decorator_args), \
                f'got {len(args)} arguments, expected {len(decorator_args)}'
            assert all([isinstance(x, t) for x, t in zip(args, decorator_args)]), \
                'unexpected types'

            return f(*args)

        return wrapper

    return decorator

@enforce_types(str, int) # decorator with arguments
def print_repeated(txt, n):
    print(txt * n)

print_repeated('Hello ', 3)
print_repeated('Hello ', 'world')
```

Python

```
@dec(argA, argB, ...)
def func(arg1, arg2, ...):
    pass
```

≡

Python

```
def func(arg1, arg2, ...):
    pass
func = dec(argA, argB, ...)(func)
```

Python shell

| Hello Hello Hello

| AssertionError: unexpected types

Example: A timer decorator

time_it.py

```
import time

def time_it(f):
    def wrapper(*args, **kwargs):
        t_start = time.time()
        result = f(*args, **kwargs)
        t_end = time.time()
        t = t_end - t_start
        print(f'{f.__name__} took {t:.2f} seconds')
        return result

    return wrapper

@time_it
def slow_function(n):
    sum_ = 0
    for x in range(n):
        sum_ += x
    print('The sum is:', sum_)

for i in range(6):
    slow_function(1_000_000 * 2 ** i)
```

Python shell

```
| The sum is: 499999500000
| slow_function took 0.27 sec
| The sum is: 1999999000000
| slow_function took 0.23 sec
| The sum is: 7999998000000
| slow_function took 0.41 sec
| The sum is: 31999996000000
| slow_function took 0.81 sec
| The sum is: 127999992000000
| slow_function took 1.52 sec
| The sum is: 511999984000000
| slow_function took 3.12 sec
```


Built-in @property

- decorator specific for class methods
- allows accessing `x.attribute()` as `x.attribute`, convenient if `attribute` does not take any arguments (also readonly)

rectangle1.py

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

# @property
    def area(self):
        return self.width * self.height
```

Python shell

```
> r = Rectangle(3, 4)
> print(r.area())
| 12
```

rectangle2.py

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height
```

Python shell

```
> r = Rectangle(3, 4)
> print(r.area)
| 12
```

Class decorators

Python

```
@dec2
@dec1
class A:
    pass
```

=

Python

```
class A:
    pass

A = dec2(dec1(A))
```

Module `dataclasses` (Since Python 3.7)

- New (and more configurable) alternative to `namedtuple`

Python shell

```
> from dataclasses import dataclass
> @dataclass # uses a decorator to add methods to the class
class Person:
    name: str # uses type annotation to define fields
    appeared: int
    height: str = 'unknown height' # field with default value
> person = Person('Donald Duck', 1934, '3 feet')
> person
| Person(name='Donald Duck', appeared=1934, height='3 feet')
> person.name
| 'Donald Duck'
> person.height = '3.5 feet'
> Person('Mickey Mouse', 1928)
| Person(name='Mickey Mouse', appeared=1928, height='unknown height')
```

docs.python.org/3/library/dataclasses.html#module-dataclasses

Raymond Hettinger - Dataclasses: The code generator to end all code generators - PyCon 2018

@functools.total_ordering (class decorator)

student.py

```
import functools

@functools.total_ordering
class Student():
    def __init__(self, name, student_id):
        self.name = name
        self.id = student_id

    def __eq__(self, other):
        return (self.name == other.name and self.id == other.id)

    def __lt__(self, other):
        my_name = ', '.join(reversed(self.name.split()))
        other_name = ', '.join(reversed(other.name.split()))
        return (my_name < other_name
                or (my_name == other_name and self.id < other.id))

donald = Student('Donald Duck', 7)
gladstone = Student('Gladstone Gander', 42)
grandma = Student('Grandma Duck', 1)
```

Automatically creates
<, <=, >, >= if at least
one of the functions
is implemented and
== is implemented

Python shell

```
> donald < grandma
| True
> grandma >= gladstone
| False
> grandma <= gladstone
| True
> donald > gladstone
| False
```

class_decorator.py

```
def add_lessequal(cls):
    '''Class decorator to add __le__ given __eq__ and __lt__.'''
    cls.__le__ = lambda self, other : self == other or self < other
    return cls # the original class cls with attribute __le__ added

def add_lessequal(cls): # alternative
    class sub_cls(cls):
        def __le__(self, other):
            return self == other or self < other
    return sub_cls # new subclass of class cls

@add_lessequal # Vector = add_lessequal(Vector)
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def _length_squared(self):
        return self.x ** 2 + self.y ** 2

    def __eq__(self, other):
        # Required, otherwise Vector(1, 2) == Vector(1, 2) is False
        return self._length_squared() == other._length_squared()

    def __lt__(self, other):
        return self._length_squared() < other._length_squared()

#     def __le__(self, other):
#         return self._length_squared() <= other._length_squared()

#     def __le__(self, other):
#         return self == other or self < other
```

Python shell

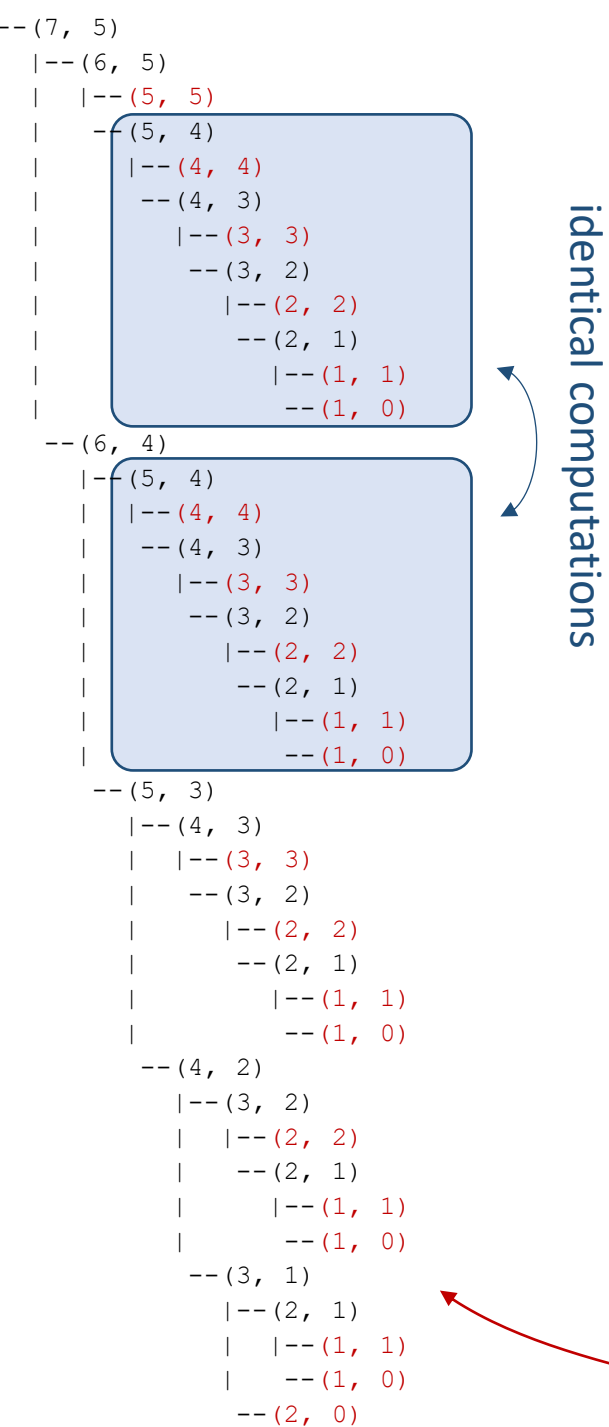
```
> u = Vector(3, 4)
> v = Vector(2, 5)
> u.__eq__(v)
| False
> u.__ne__(v)
| True # not u.__eq__(v)
> u.__lt__(v)
| True
> u.__gt__(v)
| NotImplemented # special value
> u.__le__(v)
| True # added by @add_lessequal
> u.__ge__(v)
| NotImplemented # special value
> u == v
| False
> u != v
| True
> u < v
| True
> u > v # v < u
| False
> u <= v
| True
> u >= v # v <= u
| False
```

Summary

- *@decorator_name*
- Python decorators are just syntactic sugar
- Adds functionality to a function without having to augment each call to the function or each return statement in the function
- There are decorators for functions, class methods, and classes
- There are many decorators in the Python Standard Library
- Decorators are easy to use
- ...and (slightly) harder to write

Dynamic programming

- memoization
- decorator memoized / `functools.cache`
- systematic subproblem computation



Binomial coefficient

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise} \end{cases}$$

binomial_recursive.py

```
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

recursion tree for `binomial(7, 5)`

Dynamic Programming

≡

Remember solutions already found (memoization)

- Technique sometimes applicable when running time otherwise becomes exponential
- Only applicable if stuff to be remembered is manageable

Binomial Coefficient

Dynamic programming using a dictionary

recursion tree for
`binomial(7, 5)`

```
-- (7, 5)
| -- (6, 5)
| | -- (5, 5)
| | -- (5, 4)
| | | -- (4, 4)
| | | -- (4, 3)
| | | | -- (3, 3)
| | | | -- (3, 2)
| | | | | -- (2, 2)
| | | | | -- (2, 1)
| | | | | -- (1, 1)
| | | | | -- (1, 0)
| -- (6, 4)
| | -- (5, 4)
| | -- (5, 3)
| | | -- (4, 3)
| | | -- (4, 2)
| | | | -- (3, 2)
| | | | -- (3, 1)
| | | | | -- (2, 1)
| | | | | -- (2, 0)
```

`binomial_dictionary.py`

```
answers = {} # answers[(n, k)] = binomial(n, k)

def binomial(n, k):
    if (n, k) not in answers:
        if k == 0 or k == n:
            answer = 1
        else:
            answer = binomial(n - 1, k) + binomial(n - 1, k - 1)
        answers[(n, k)] = answer
    return answers[(n, k)]
```

`Python shell`

```
> binomial(6, 3)
| 20
> answers
| {(3, 3): 1, (2, 2): 1, (1, 1): 1, (1, 0): 1, (2, 1): 2, (3, 2): 3, (4, 3): 4, (2, 0): 1, (3, 1): 3, (4, 2): 6, (5, 3): 10, (3, 0): 1, (4, 1): 4, (5, 2): 10, (6, 3): 20}
```

- Use a dictionary `answers` to store already computed values
- reuse value stored in dictionary `answers`

Question – What is the order of the size of the dictionary
answers after calling `binomial(n, k)` ?

`binomial_dictionary.py`

```
answers = {} # answers[(n, k)] = binomial(n, k)

def binomial(n, k):
    if (n, k) not in answers:
        if k == 0 or k == n:
            answer = 1
        else:
            answer = binomial(n - 1, k) + binomial(n - 1, k - 1)
        answers[(n, k)] = answer
    return answers[(n, k)]
```

- a) $\max(n, k)$
- b) $n + k$
-  c) $n * k$
- d) n^k
- e) k^n
- f) Don't know

Binomial Coefficient

Dynamic programming using decorator

- Use a decorator (@memoize) that implements the functionality of remembering the results of previous function calls

binomial_decorator.py

```
def memoize(f):  
    # answers[args] = f(*args)  
    answers = {}  
    def wrapper(*args):  
        if args not in answers:  
            answers[args] = f(*args)  
        return answers[args]  
    return wrapper
```

```
@memoize  
def binomial(n, k):  
    if k == 0 or k == n:  
        return 1  
    else:  
        return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

binomial_decorator_trace.py	Python shell (without @memoize)	Python shell (with @memoize)
<pre> def trace(f): # decorator to trace recursive calls indent = 0 def wrapper(*args): nonlocal indent spaces = ' ' * indent arg_str = ', '.join(map(repr, args)) print(spaces + f'{f.__name__}({arg_str})') indent += 1 result = f(*args) indent -= 1 print(spaces + f'> {result}') return result return wrapper def memoize(f): answers = {} def wrapper(*args): if args not in answers: answers[args] = f(*args) return answers[args] wrapper.__name__ = f.__name__ + '_memoize' return wrapper @trace @memoize def binomial(n, k): if k == 0 or k == n: return 1 return binomial(n - 1, k) + binomial(n-1, k-1) print(binomial(5, 2)) </pre>	<pre> binomial(5, 2) binomial(4, 2) binomial(3, 2) binomial(2, 2) > 1 binomial(2, 1) binomial(1, 1) > 1 binomial(1, 0) > 1 > 2 > 3 binomial(3, 1) binomial(2, 1) binomial(1, 1) > 1 binomial(1, 0) > 1 > 2 > 3 binomial(2, 0) > 1 > 3 > 6 binomial(4, 1) binomial(3, 1) binomial(2, 1) binomial(1, 1) > 1 binomial(1, 0) > 1 > 2 binomial(2, 0) > 1 > 3 > 4 binomial(3, 0) > 1 > 1 10 </pre>	<pre> binomial_memoize(5, 2) binomial_memoize(4, 2) binomial_memoize(3, 2) binomial_memoize(2, 2) > 1 binomial_memoize(2, 1) binomial_memoize(1, 1) > 1 binomial_memoize(1, 0) > 1 > 2 > 3 binomial_memoize(3, 1) binomial_memoize(2, 1) > 2 binomial_memoize(2, 0) > 1 > 3 > 6 binomial_memoize(4, 1) binomial_memoize(3, 1) > 3 binomial_memoize(3, 0) > 1 > 4 > 10 10 </pre> without assigning <code>wrapper.__name__</code> the name shown would be <code>wrapper</code> saved recursive calls when using memoization

Dynamic programming using cache decorator

```
binomial_cache.py

from functools import cache

@cache
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

- The decorators `@cache` (since Python 3.9) and `@lru_cache(maxsize=None)` in the standard library `functools` supports the same as the decorator `@memoize`
- By default `@lru_cache` at most remembers (caches) 128 previous function calls, always evicting **Least Recently Used** entries from its dictionary
- `functools.cache` can have problems when using `sys.setrecursionlimit` (e.g. with Python 3.13 on Windows) 

Subset sum using dynamic programming

- In the subset sum problem (Exercise 13.4) we are given a number x and a list of numbers $\mathbb{L}$, and want to determine if a subset of $\mathbb{L}$ has sum x

$$\mathbb{L} = [3, 7, 2, 11, 13, 4, 8] \quad x = 22 = 7 + 11 + 4$$

- Let $S(v, k)$ denote if it is possible to achieve value v with a subset of $\mathbb{L}[:k]$, i.e. $S(v, k) = \text{True}$ if and only if a subset of the first k values in $\mathbb{L}$ has sum v
- $S(v, k)$ can be computed from the recurrence

$$S(v, k) = \begin{cases} \text{True} & \text{if } k = 0 \text{ and } v = 0 \\ \text{False} & \text{if } k = 0 \text{ and } v \neq 0 \\ S(v, k-1) \text{ or } S(v - \mathbb{L}[k-1], k-1) & \text{otherwise} \end{cases}$$

Subset sum using dynamic programming

subset_sum_dp.py

```
def subset_sum(x, L):  
    @memoize  
    def solve(value, k):  
        if k == 0:  
            return value == 0  
        return solve(value, k - 1) or solve(value - L[k - 1], k - 1)  
    return solve(x, len(L))
```

Python shell

```
> subset_sum(11, [2, 3, 8, 11, -1])  
| True  
> subset_sum(6, [2, 3, 8, 11, -1])  
| False
```

$$S(v, k) = \begin{cases} \text{True} & \text{if } k = 0 \text{ and } v = 0 \\ \text{False} & \text{if } k = 0 \text{ and } v \neq 0 \\ S(v, k-1) \text{ or } S(v - L[k-1], k-1) & \text{otherwise} \end{cases}$$

Question – What is a bound on the size order of the memoization table if all values are possitive integers?

subset_sum_dp.py

```
def subset_sum(x, L):  
    @memoize  
    def solve(value, k):  
        if k == 0:  
            return value == 0  
        return solve(value, k-1) or solve(value - L[k-1], k-1)  
    return solve(x, len(L))
```

Python shell

```
> subset_sum(11, [2, 3, 8, 11, -1])  
| True  
> subset_sum(6, [2, 3, 8, 11, -1])  
| False
```

a) `len(L)`

b) `sum(L)`

c) `x`

 d) $2^{\text{len}(L)}$

e) `len(L)`

 f) `len(L) * sum(L)`

g) Don't know

Subset sum using dynamic programming

subset_sum_dp.py

```
def subset_sum_solution(x, L):  
    @memoize  
    def solve(value, k):  
        if k == 0:  
            if value == 0:  
                return []  
            else:  
                return None  
        solution = solve(value, k - 1)  
        if solution != None:  
            return solution  
        solution = solve(value - L[k - 1], k - 1)  
        if solution != None:  
            return solution + [L[k - 1]]  
        return None  
    return solve(x, len(L))
```

Python shell

```
> subset_sum_solution(11, [2, 3, 8, 11, -1])  
| [3, 8]  
> subset_sum_solution(6, [2, 3, 8, 11, -1])  
| None
```

Knapsack problem

	Volume	Value
0	3	6
1	3	7
2	2	8
3	5	9

- Given a **knapsack** with volume **capacity C**, and set of **objects** with different **volumes and value**
- **Objective:** Find a subset of the objects that fits in the knapsack (sum of volume $\leq$ capacity) and has maximal value
- **Example:** If $C = 5$ and the volume and weights are given by the table, then the maximal value 15 can be achieved by the 2nd and 3rd object
- Let $V(c, k)$ denote the **maximum value achievable by a subset of the first k objects within capacity c**

$$V(c, k) = \begin{cases} 0 & \text{if } k = 0 \\ V(c, k - 1) & \text{volume}[k-1] > c \\ \max\{V(c, k - 1), \text{value}[k - 1] + V(c - \text{volume}[k - 1], k - 1)\} & \text{otherwise} \end{cases}$$

Knapsack – maximum value

knapsack.py

```
def knapsack_value(volume, value, capacity):  
    @memoize  
    def solve(c, k): # solve with capacity c and objects 0..k-1  
        if k == 0: # no objects to put in knapsack  
            return 0  
        v = solve(c, k - 1) # try without object k-1  
        if volume[k - 1] <= c: # try also with object k-1 if space  
            v = max(v, value[k - 1] + solve(c - volume[k - 1], k - 1))  
        return v  
    return solve(capacity, len(volume))
```

Python shell

```
> volumes = [3, 3, 2, 5]  
> values = [6, 7, 8, 9]  
> knapsack_value(volumes, values, 5)  
| 15
```

Knapsack – maximum value and objects

knapsack.py

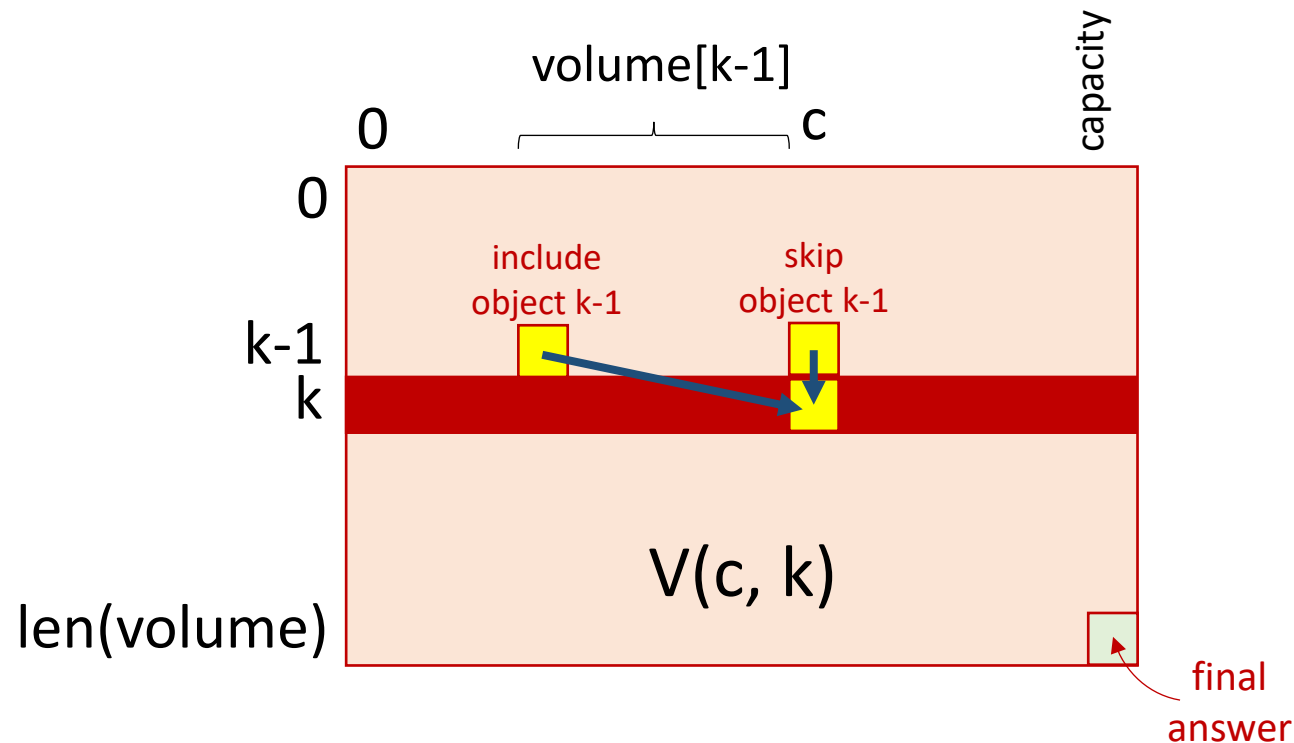
```
def knapsack(volume, value, capacity):  
    @memoize  
    def solve(c, k): # solve with capacity c and objects 0..k-1  
        if k == 0: # no objects to put in knapsack  
            return 0, []  
        v, solution = solve(c, k - 1) # try without object k-1  
        if volume[k - 1] <= c: # try also with object k-1 if space  
            v2, sol2 = solve(c - volume[k - 1], k - 1)  
            v2 = v2 + value[k - 1]  
            if v2 > v:  
                v = v2  
                solution = sol2 + [k - 1]  
        return v, solution  
    return solve(capacity, len(volume))
```

Python shell

```
> volumes = [3, 3, 2, 5]  
> values = [6, 7, 8, 9]  
> knapsack(volumes, values, 5)  
| (15, [1, 2])
```

Knapsack - Table

$$V(c, k) = \begin{cases} 0 & \text{if } k = 0 \\ V(c, k - 1) & \text{value}[k-1] > c \\ \max\{V(c, k - 1), \text{value}[k - 1] + V(c - \text{volume}[k - 1], k - 1)\} & \text{otherwise} \end{cases}$$



- systematic fill out table
- only need to remember two rows

Knapsack – Systematic table fill out

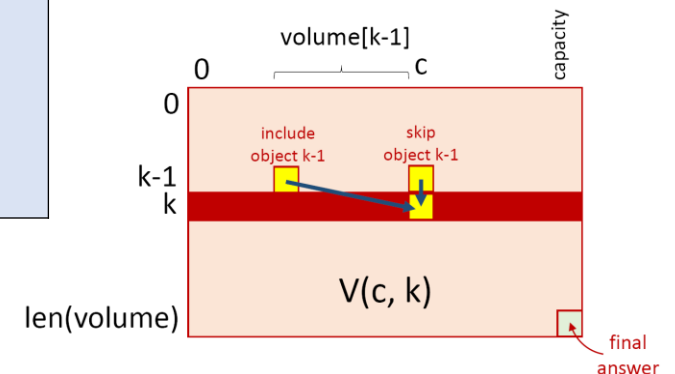
knapsack_systematic.py

```
def knapsack(volume, value, capacity):  
    ① solutions = [(0, [])] * (capacity + 1)  
    ② for obj in range(len(volume)): ⑤  
        ④ for c in reversed(range(volume[obj], capacity + 1)):  
            prev_v, prev_solution = solutions[c - volume[obj]]  
            v = value[obj] + prev_v  
            if solutions[c][0] < v:  
                ③ solutions[c] = v, prev_solution + [obj]  
  
    return solutions[capacity]
```

Python shell

```
> volumes = [3, 3, 2, 5]  
> values = [6, 7, 8, 9]  
> knapsack(volumes, values, 5)  
| (15, [1, 2])
```

- ① base case $k = 0$
- ② consider each object
- ③ $\text{solutions}[c:]$ current row
 $\text{solutions}[:c]$ previous row
- ④ compute next row right-to-left
- ⑤ $\text{solutions}[:\text{volume}[\text{obj}]]$
unchanged from previous row



Summary

- Dynamic programming is a general approach for recursive problems where one tries to avoid recomputing the same expressions repeatedly
- **Solution 1: Memoization**
 - add dictionary to function to remember previous results
 - decorate with a `@memoize` decorator
- **Solution 2: Systematic table fill out**
 - can need to compute more values than when using memoization
 - can discard results not needed any longer (reduced memory usage)

Coding competitions and online judges

If you like to practice your coding skills, there are many online “judges” with numerous exercises and where you can upload and test your solutions.

- [Project Euler](#)
- [Kattis](#)
- [CodeForces](#)
- [Topcoder](#)

Visualization and optimization

- Matplotlib
- Jupyter
- `scipy.optimize.minimize`



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 shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

pip install matplotlib

matplotlib.org

Plot

pyplot module $\approx$ MATLAB-like plotting framework

matplotlib-simple.py

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3], [5, 2, 7], 'bo:')

plt.show()
```

add plot
to figure

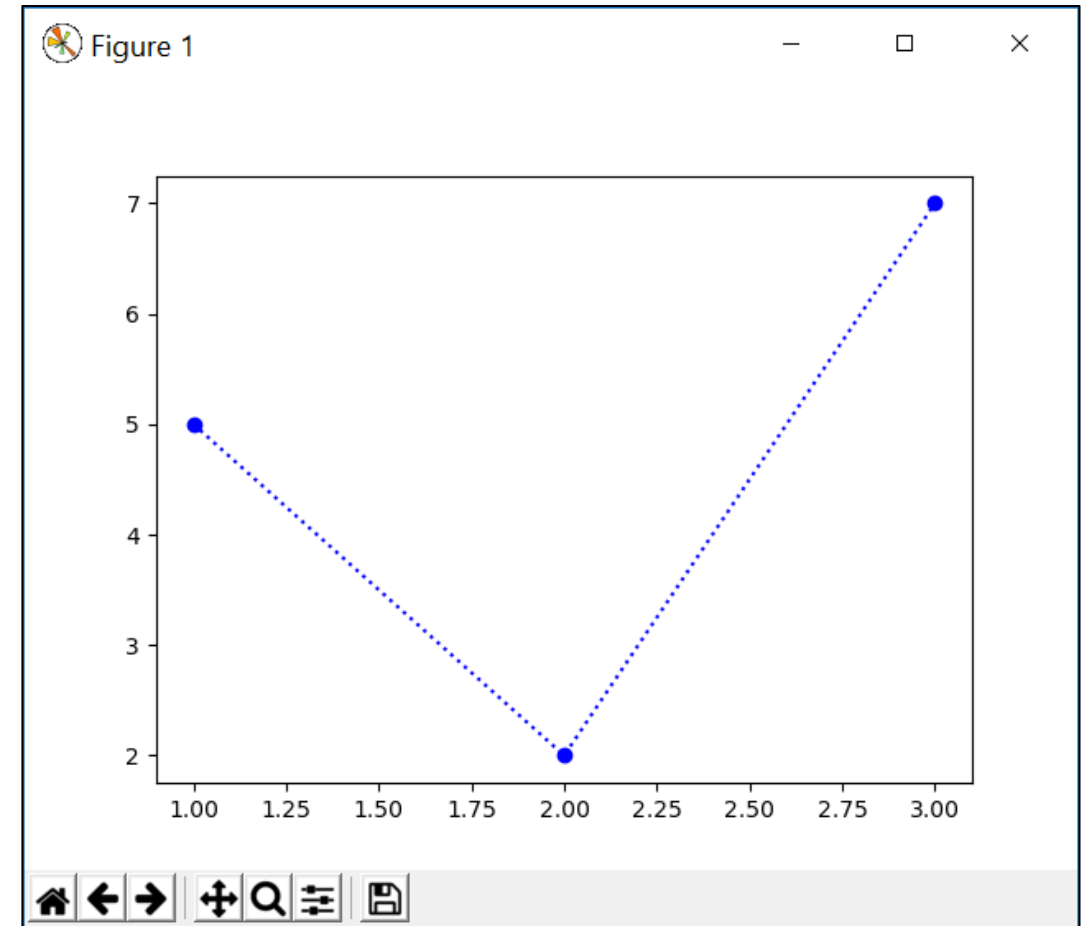
figure is first shown
when show is called

x coordinates

y coordinates

format
string

Colors	Line styles	Marker styles
b 	- 	.  2  + 
g 	-- 	,  3  x 
r 	-. 	o  4  D 
c 	:	v  s  d 
m 		^  p  
y 		<  *  - 
k 		>  h 
w 		1  H 



save current view as picture

adjust margins

zoom rectangle

pan and zoom

navigate view history

reset view

Plot – some keyword arguments

matplotlib-plot.py

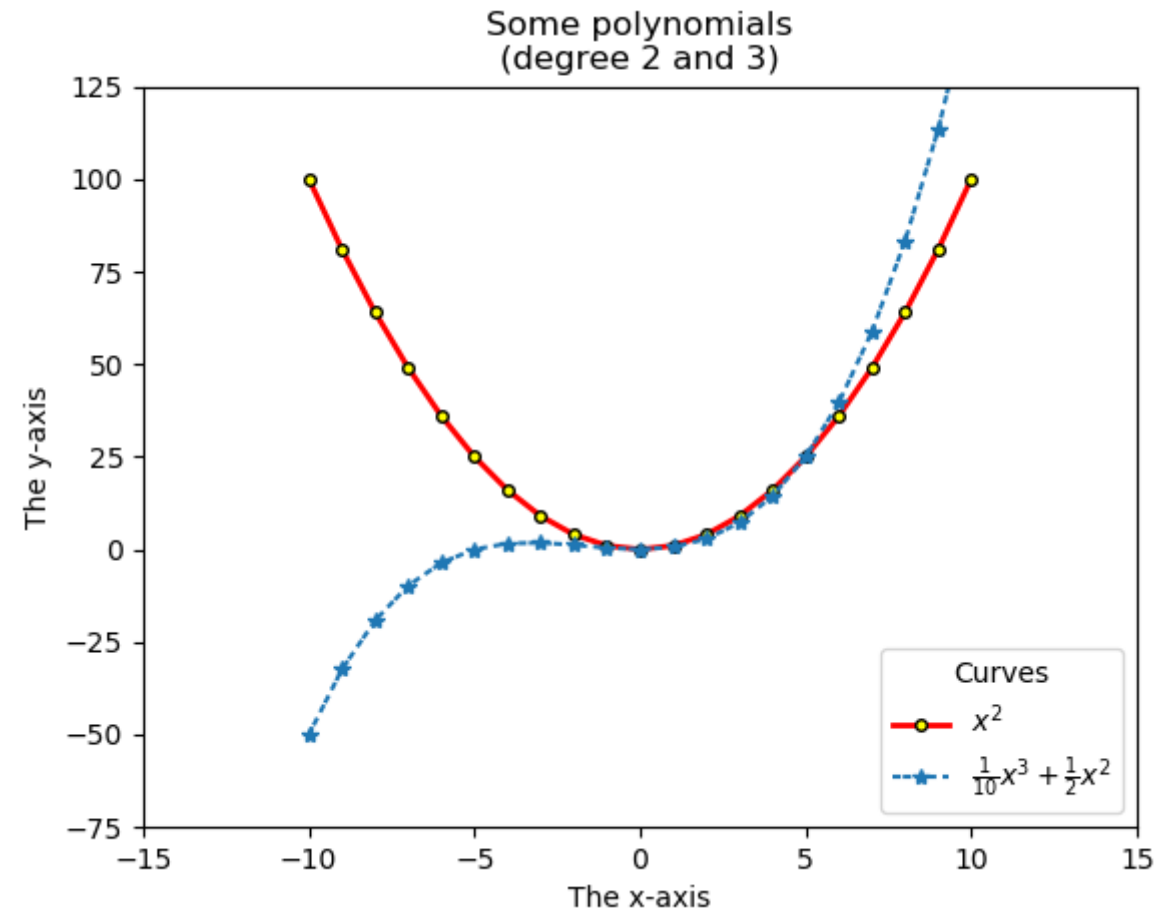
```
import matplotlib.pyplot as plt

X = range(-10, 11)
Y1 = [x ** 2 for x in X]
Y2 = [x ** 3 / 10 + x ** 2 / 2 for x in X]

plt.plot(X, Y1, color='red', label='$x^2$',
         linestyle='-', linewidth=2,
         marker='o', markersize=4,
         markeredgewidth=1,
         markeredgecolor='black',
         markerfacecolor='yellow')

plt.plot(X, Y2, '*', dashes=(2, 0.5, 2, 1.5),
         label=r'$\frac{1}{10}x^3+\frac{1}{2}x^2$')
plt.xlim(-15, 15)
plt.ylim(-75, 125)
plt.title('Some polynomials\n(degree 2 and 3)')
plt.xlabel('The x-axis')
plt.ylabel('The y-axis')
plt.legend(title='Curves')

plt.show() # finally show figure
```



matplotlib.org/api/ as gen/matplotlib.pyplot.plot.html

Colors: matplotlib.org/gallery/color/named_colors.html

Scatter (points with individual size and color)

colorbar
(of most recently
used colormap)

matplotlib-scatter.py

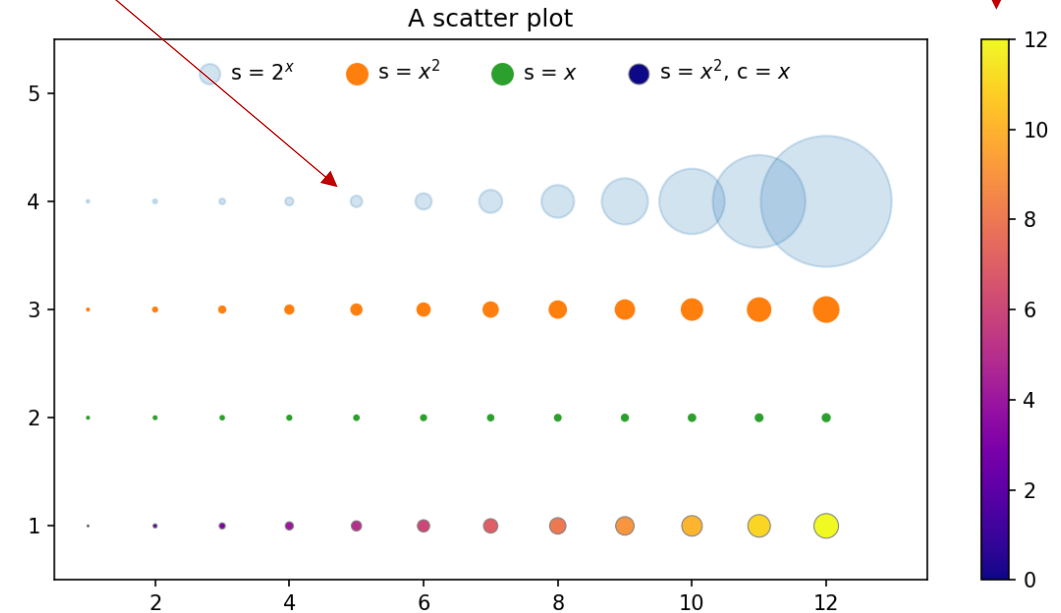
```
import matplotlib.pyplot as plt

n = 13
X = range(n)
S = [x ** 2 for x in X]
E = [2 ** x for x in X]

plt.scatter(X, [4] * n, s=E, label='s =  $2^x$ ', alpha=.2)
plt.scatter(X, [3] * n, s=S, label='s =  $x^2$ ')
plt.scatter(X, [2] * n, s=X, label='s =  $x$ ')
plt.scatter(X, [1] * n, s=S, c=X, cmap='plasma',
            label='s =  $x^2$ , c =  $x$ ',
            edgecolors='gray', linewidth=0.5)
plt.colorbar()

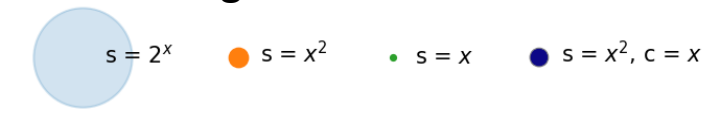
plt.ylim(0.5, 5.5)
plt.xlim(0.5, 13.5)
plt.title('A scatter plot')
legend = plt.legend(loc='upper center', frameon=False,
                  ncol=4, handletextpad=0)
for handle in legend.legend_handles:
    handle.set_sizes([100]) # fix marker size in legend
plt.show()
```

transparency



colormap (predefined)
color of each point
size $\approx$ area of each point
point boundary width
point boundary color

Legend without fix



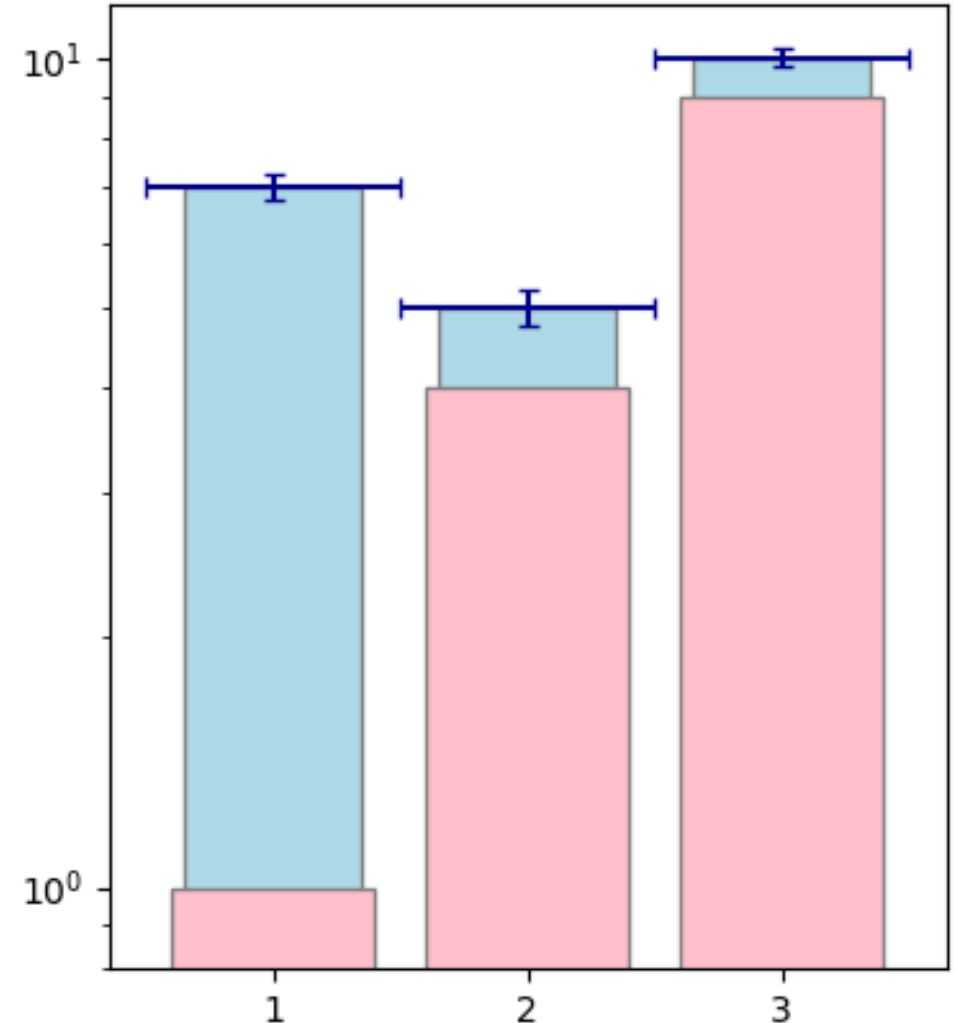
manual placement of legend box (default automatic); remove frame; place legends in 4 columns (default 1); reduce space between marks and label

matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html
matplotlib.org/tutorials/colors/colormaps.html

Bars

matplotlib-bars.py

```
import matplotlib.pyplot as plt
x = [1, 2, 3]
y = [7, 5, 10]
plt.bar(x, y,
        color='lightblue', # bar background color
        linewidth=1,       # bar boundary width
        edgecolor='gray',  # bar boundary color
        tick_label=x,     # ticks on x-axis
        width=0.7,        # width, default 0.8
        yerr=0.25,         # Error bar: y length
        xerr=0.5,          # x length
        capsize=3,         # capsize in points
        ecolor='darkblue', # error bar color
        log=True)          # y-axis log scale
plt.bar(x, [v**2 for v in x],
        color='pink',
        linewidth=1,
        edgecolor='gray')
plt.show()
```



Histogram

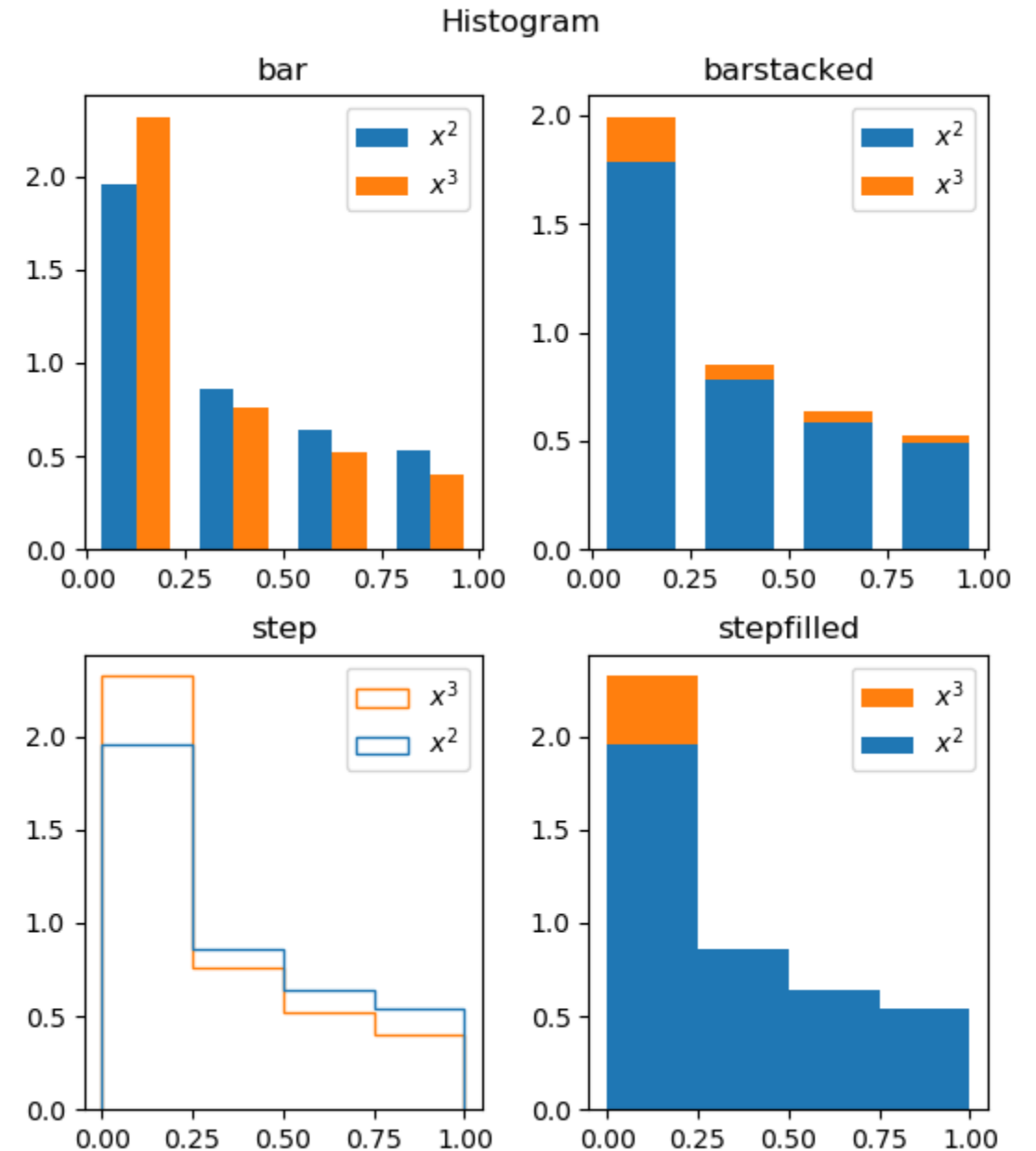
matplotlib-histogram.py

```
import matplotlib.pyplot as plt
from random import random

values1 = [random()*2 for _ in range(1000)]
values2 = [random()*3 for _ in range(100)]
bins = [0.0, 0.25, 0.5, 0.75, 1.0]

for i, ht in enumerate(
    ['bar', 'barstacked', 'step', 'stepfilled'],
    start=1):
    plt.subplot(2, 2, i) # start new plot
    plt.hist([values1, values2], # data sets
             bins, # bucket boundaries
             histtype=ht, # default ht='bar'
             rwidth=0.7, # fraction of bucket width
             label=['$x^2$', '$x^3$'], # labels
             density=True) # norm. prob. density
    plt.title(ht) # plot title
    plt.xticks(bins) # ticks on x-axis
    plt.legend()

plt.suptitle('Histogram') # figure title
plt.show()
```

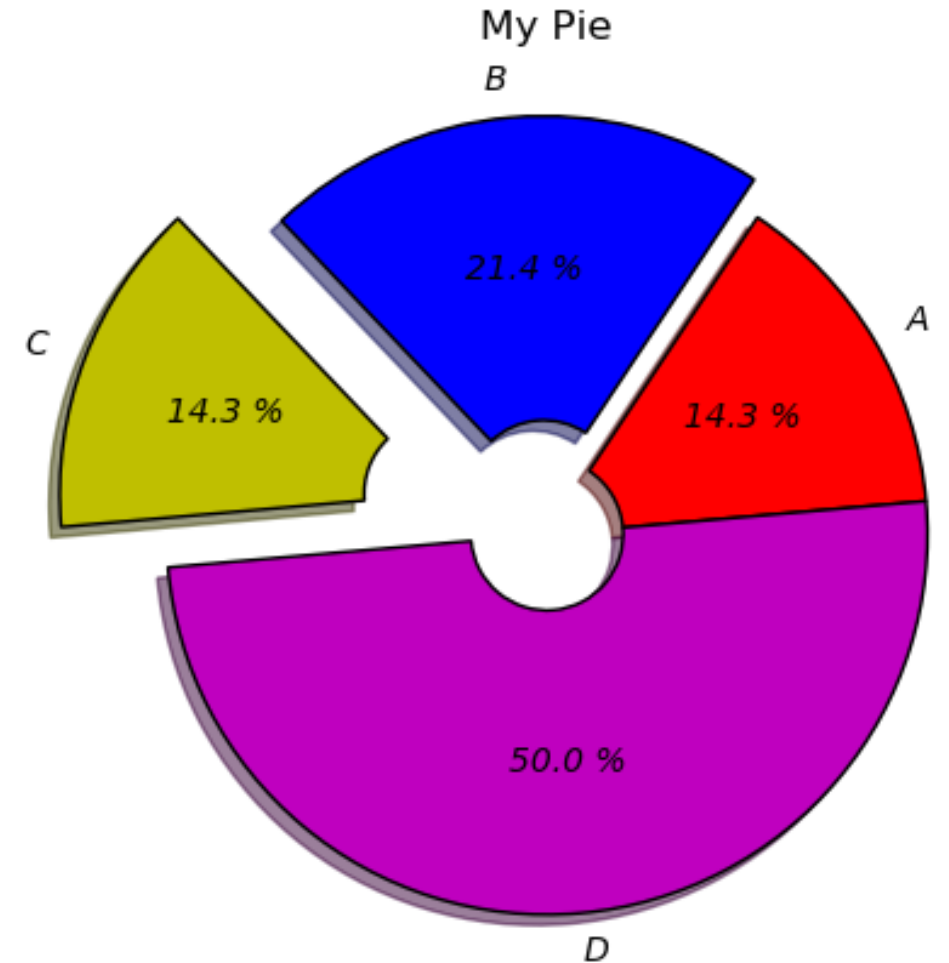


Pie

matplotlib-pie.py

```
import matplotlib.pyplot as plt
plt.title('My Pie')
plt.pie([2, 3, 2, 7],          # relative wedge sizes
        labels=['A', 'B', 'C', 'D'],
        colors=['r', 'b', 'y', 'm'],
        explode=(0, 0.1, 0.3, 0), # radius fraction
        startangle=5,           # angle above horizontal
        counterclock=True,      # default True
        rotatelabels=False,     # default False
        shadow=True,            # default False
        textprops=dict(        # text properties, dict
            color='black',      # text color
            style='italic'),     # text style
        wedgeprops=dict(       # wedge properties, dict
            width=0.8,          # width (missing center)
            linewidth=1,        # wedge boundary width
            edgecolor='black'),  # boundary color
        autopct='%1.1f %%')    # percent formatting

plt.show()
```



Customizing Pie shadows

- Need to do it manually on each pie using `matplotlib.patches.Shadow`

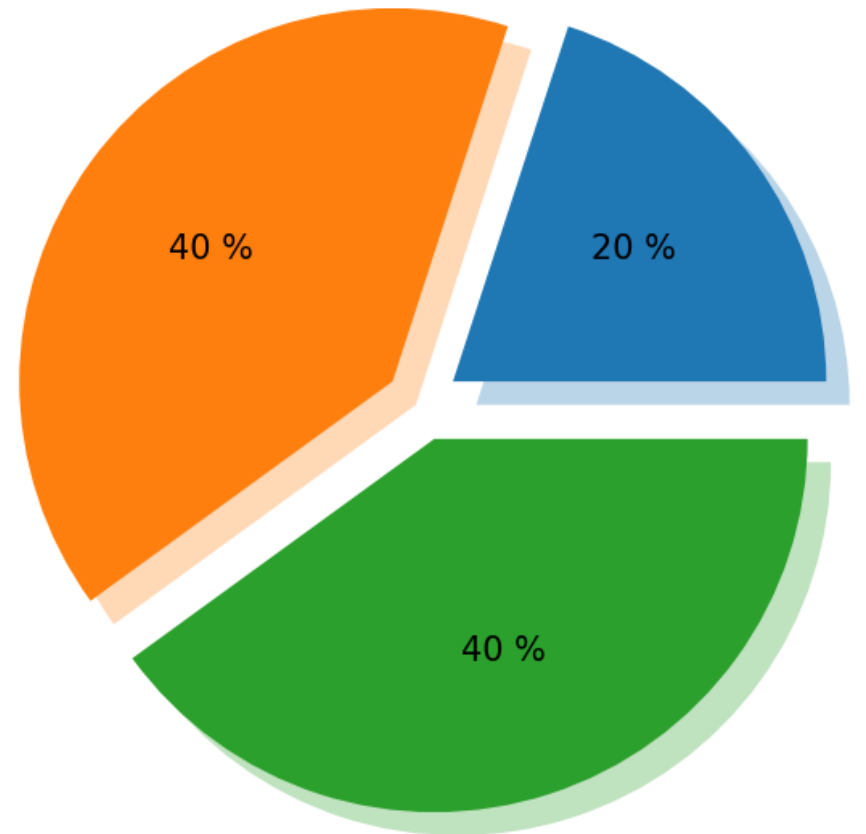
`matplotlib-pie-shadow.py`

```
import matplotlib.pyplot as plt
from matplotlib.patches import Shadow

patches, texts, autotexts = plt.pie(
    [1, 2, 2],
    explode=(0.1, 0.1, 0.1),
    autopct='%1.0f %%'
)

for pie in patches:
    pie_shadow = Shadow(
        pie, 0.03, -0.03, # patch, x-offset, y-offset
        alpha=0.3,        # shadow transparency
        edgecolor=None,    # shadow edge color
        facecolor=pie._facecolor # shadow fill color
    )
    plt.gca().add_patch(pie_shadow)

plt.show()
```



Stackplot

matplotlib-stackplot.py

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y1 = [1, 2, 3, 4]
y2 = [2, 3, 1, 4]
y3 = [2, 4, 1, 3]

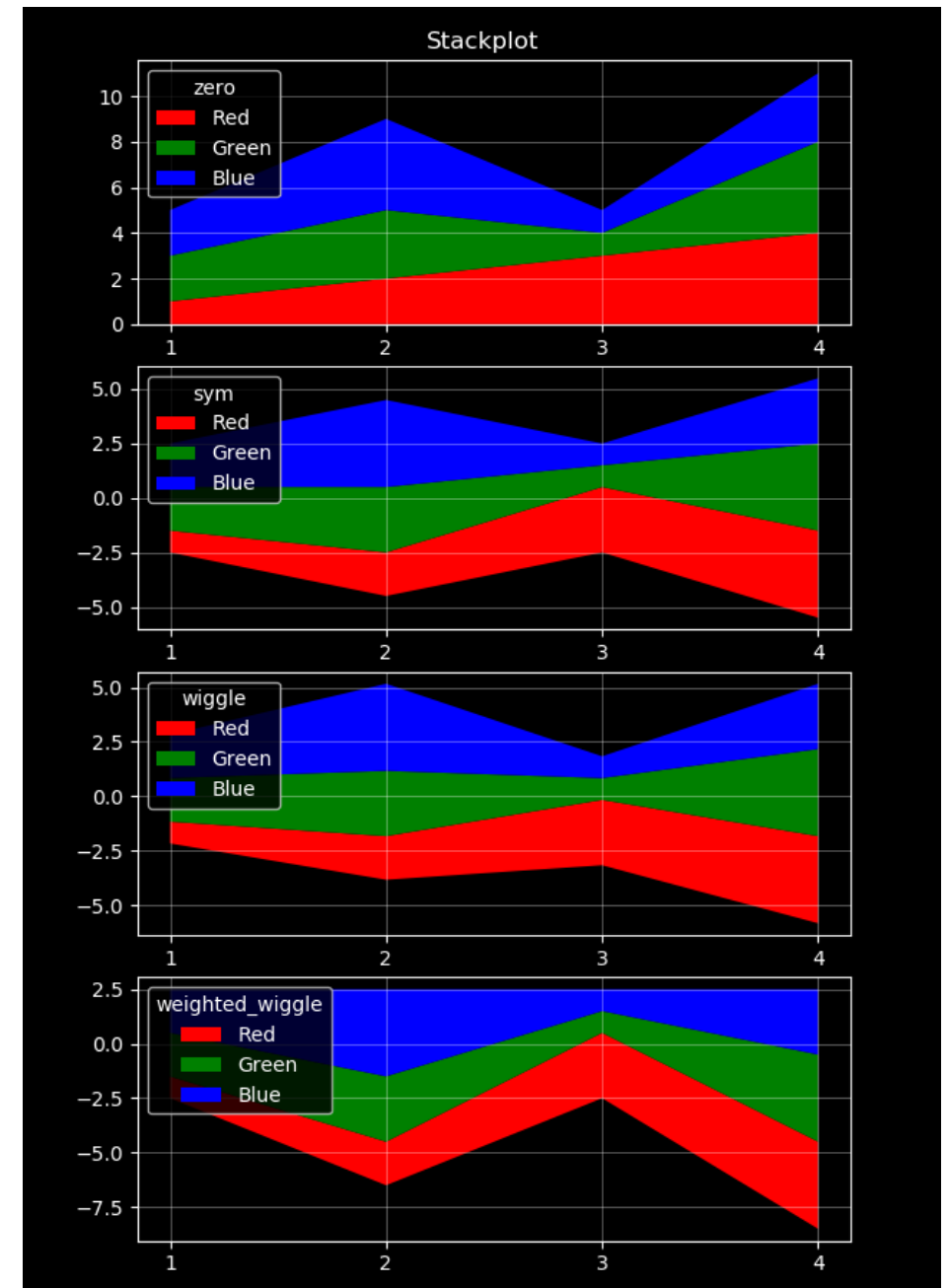
plt.style.use('dark_background')
for i, base in enumerate(
    ['zero', 'sym', 'wiggle', 'weighted_wiggle'],
    start=1):
    plt.subplot(4, 1, i)
    plt.stackplot(x, y1, y2, y3,
                  colors=['r', 'g', 'b'],
                  labels=['Red', 'Green', 'Blue'],
                  baseline=base)

    plt.grid(axis='both', # 'x', 'y', or 'both'
            linewidth=0.5, linestyle='-', alpha=0.5)
    plt.legend(title=base, loc='upper left')
    plt.xticks(x) # a tick for each value in x

plt.suptitle('Stackplot')
plt.show()
```

Stacked Graphs – Geometry & Aesthetics
Lee Byron & Martin Wattenberg, 2008

To list all available styles:
`print(plt.style.available)`



```

import matplotlib.pyplot as plt
from math import pi, sin

x_min, x_max, n = 0, 2 * pi, 100
x = [x_min + (x_max - x_min) * i / n for i in range(n + 1)]
y = [sin(v) for v in x]

ax1 = plt.subplot(2, 3, 1) # 2 rows, 3 columns
ax1.label_outer()         # removes x-axis labels
plt.xlim(-pi, 3 * pi)     # increase x-axis range
plt.plot(x, y, 'r-')
plt.title('Plot A')

ax2 = plt.subplot(2, 3, 2)
ax2.label_outer()         # removes x- and y-axis labels
plt.xlim(-2 * pi, 4 * pi) # increase x-axis range
plt.plot(x, y, 'g-')
plt.title('Plot B')

ax3 = plt.subplot(2, 3, 3, frameon=False) # remove frame
ax3.set_xticks([])                     # remove x-axis ticks & labels
ax3.set_yticks([])                     # remove y-axis ticks & labels
plt.plot(x, y, 'b--')
plt.title('No frame')

ax4 = plt.subplot(2, 3, 4, sharex=ax1) # share x-axis range
plt.ylim(-2, 2)                        # increase y-axis range
plt.plot(x, y, 'm-')
plt.title('Plot C')

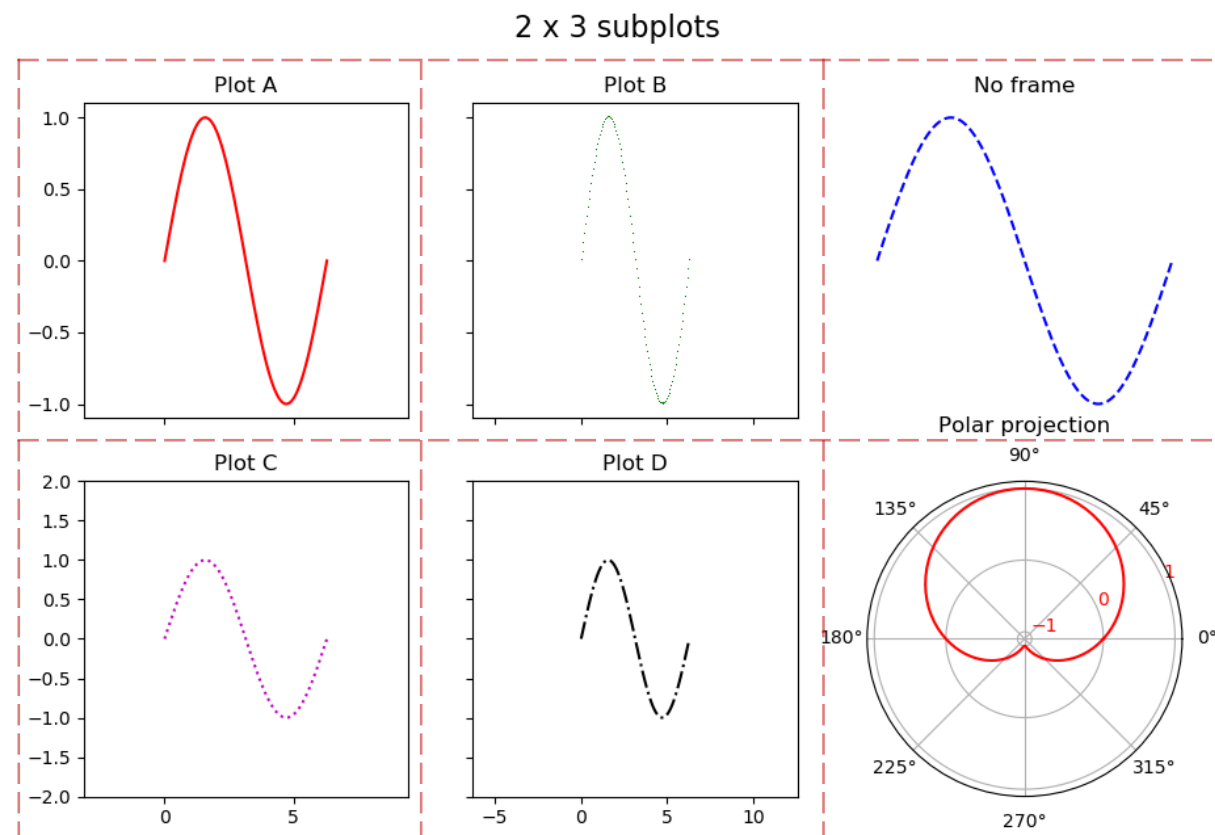
ax5 = plt.subplot(2, 3, 5, sharex=ax2, sharey=ax4) # share ranges
ax5.set_xticks(range(-5, 15, 5)) # specific x-ticks & x-labels
ax5.label_outer()                 # removes y-axis labels
plt.plot(x, y, 'k-')
plt.title('Plot D')

ax6 = plt.subplot(2, 3, 6, projection='polar') # polar projection
ax6.set_yticks([-1, 0, 1])                    # y-labels
ax6.tick_params(axis='y', labelcolor='red')    # color of y-labels
plt.plot(x, y, 'r')
plt.title('Polar projection\n') # \n to avoid overlap with 90°
plt.suptitle('2 x 3 subplots', fontsize=16)
plt.show()

```

Subplot

(2 rows, 3 columns)



- Subplots are numbered 1..6 row-by-row, starting top-left
- subplot returns an `axes` to access the plot in the figure

Subplots

matplotlib-subplots.py

```
import matplotlib.pyplot as plt
from math import pi, sin, cos

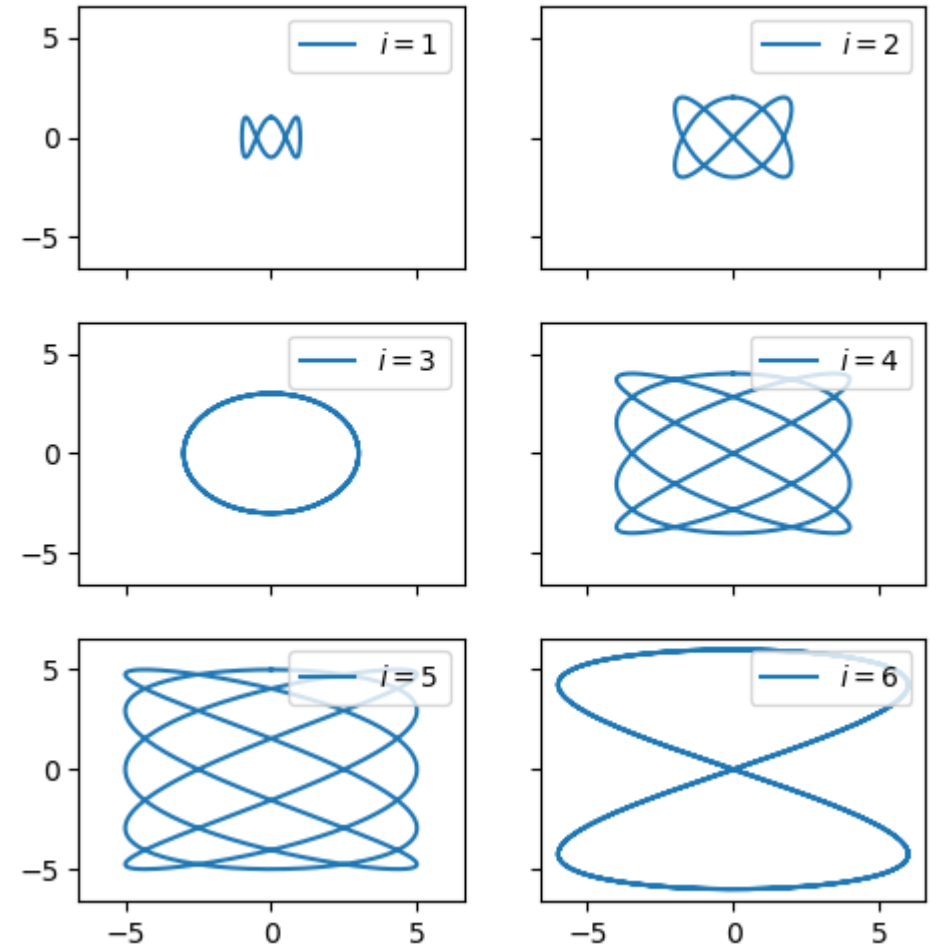
times = [2 * pi * t / 1000 for t in range(1001)]

fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6)) = \
    plt.subplots(3, 2, sharex=True, sharey=True)

for i, ax in enumerate([ax1, ax2, ax3, ax4, ax5, ax6],
                        start=1):
    x = [i * sin(i * t) for t in times]
    y = [i * cos(3 * t) for t in times]
    ax.plot(x, y, label=f'$i = {i}$') # plot to axes
    ax.legend(loc='upper right')      # axes legend
fig.suptitle('subplots', fontsize=16) # figure title
plt.show()
```

create 6 axes in 3 rows with 2 columns
share the x- and y-axis ranges (automatically
applies label_outer to created axes)
returns a pair (figure, axes)

subplots



matplotlib-subplot2grid.py

```
import matplotlib.pyplot as plt
import math

x_min, x_max, n = 0, 2 * math.pi, 20

x = [x_min + (x_max - x_min) * i / n
      for i in range(n + 1)]
y = [math.sin(v) for v in x]

plt.subplot2grid((5, 5), (0, 0),
                 rowspan=3, colspan=3)
plt.fill_between(x, 0.0, y,
                 alpha=0.25, color='r')
plt.plot(x, y, 'r-')
plt.title('Plot A')

plt.subplot2grid((5, 5), (0, 3),
                 rowspan=2, colspan=2)
plt.plot(x, y, 'g.')
plt.title('Plot B')

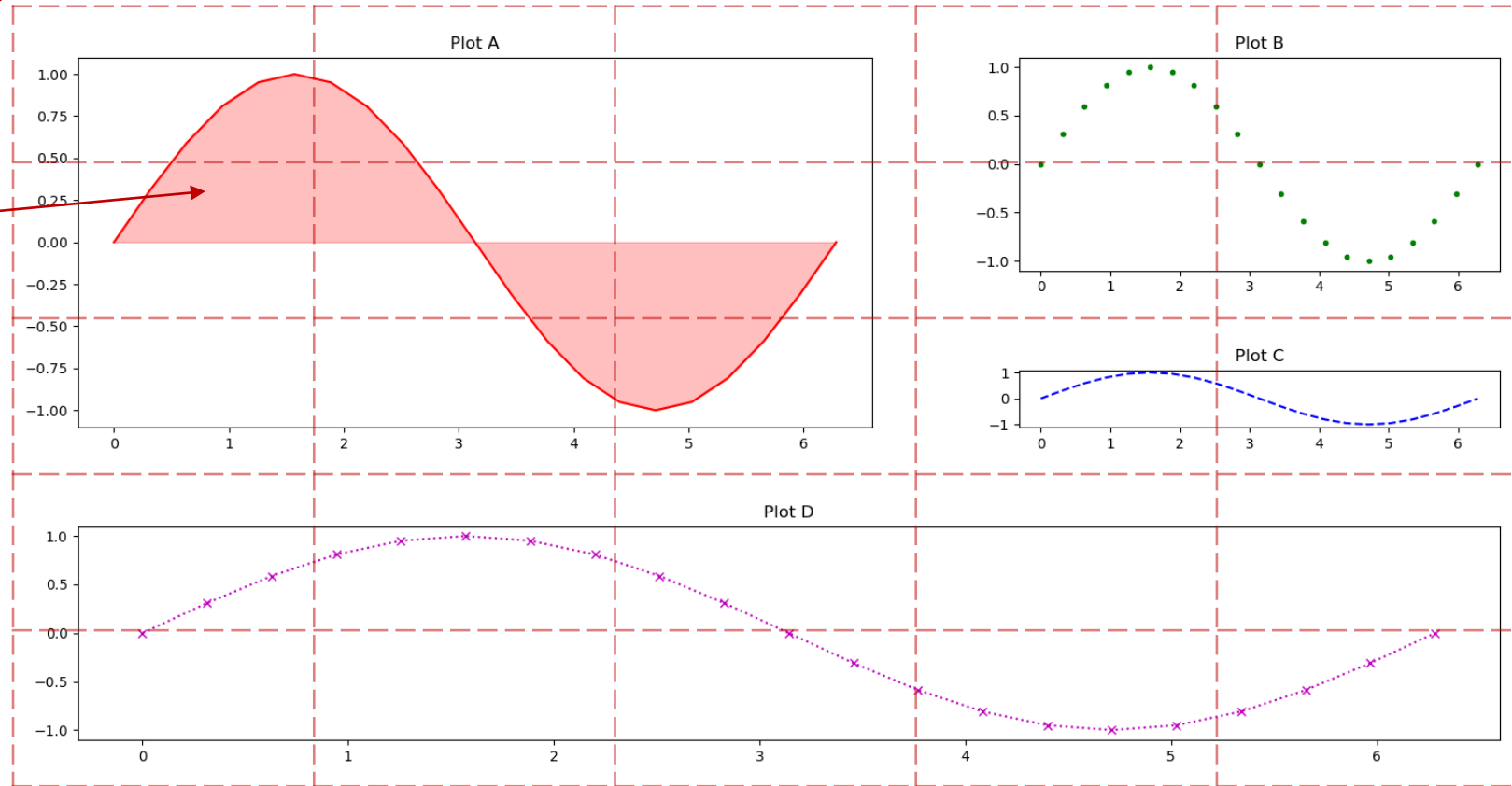
plt.subplot2grid((5, 5), (2, 3),
                 rowspan=1, colspan=2)
plt.plot(x, y, 'b--')
plt.title('Plot C')

plt.subplot2grid((5, 5), (3, 0),
                 rowspan=2, colspan=5)
plt.plot(x, y, 'mx:')
plt.title('Plot D')

plt.tight_layout() # adjust padding
plt.show()
```

subplot2grid (5 x 5)

upper left corner (row, column)



```

import matplotlib.pyplot as plt

x = [i / 10 for i in range(1, 101)]

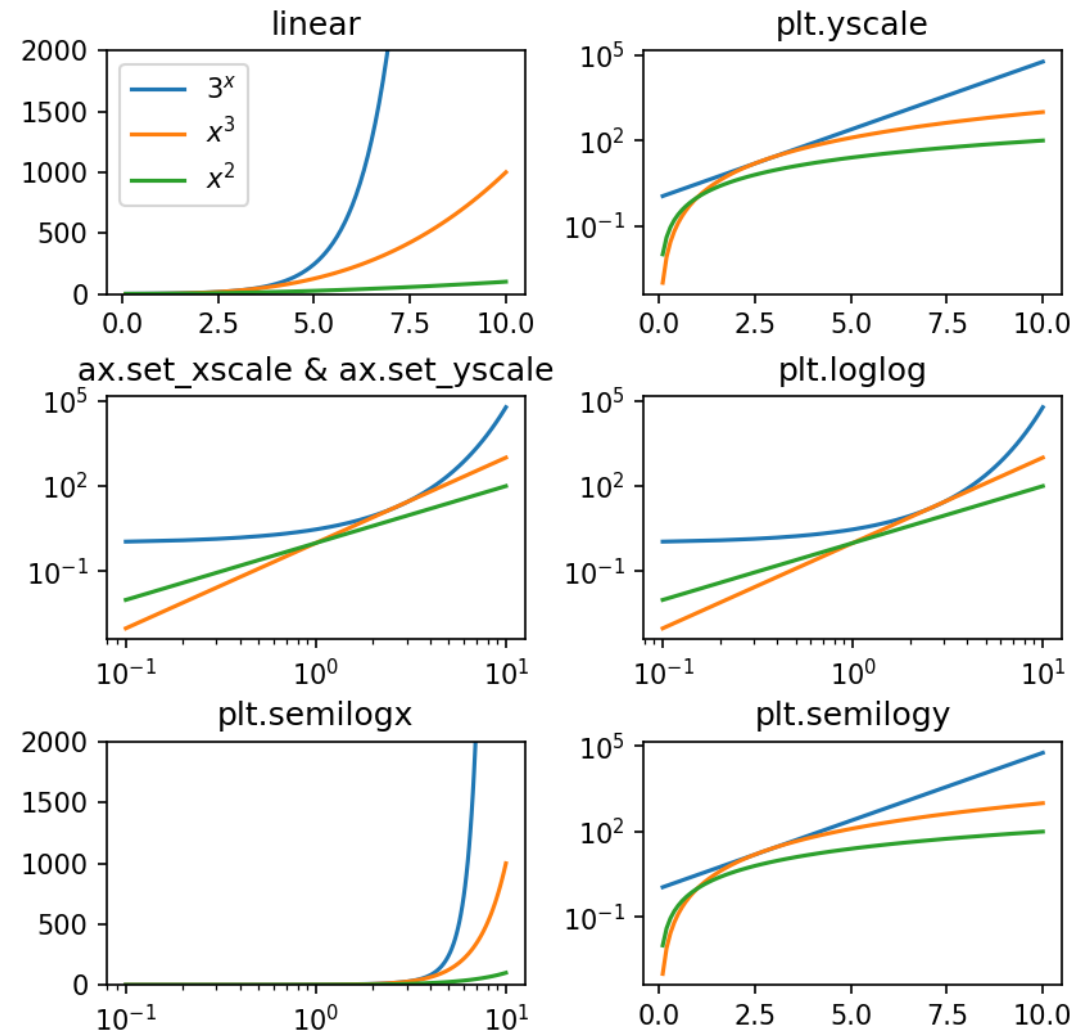
y1 = [i ** 2 for i in x]
y2 = [i ** 3 for i in x]
y3 = [3 ** i for i in x]

for i in range(1, 7):
    ax = plt.subplot(3, 2, i)
    plt.plot(x, y3, label='$3^x$')
    plt.plot(x, y2, label='$x^3$')
    plt.plot(x, y1, label='$x^2$')
    match i:
        case 1:
            plt.ylim(0, 2000)
            plt.xscale('linear') # default
            plt.yscale('linear') # default
            plt.legend()
            plt.title('linear')
        case 2:
            plt.yscale('log')
            plt.title('plt.yscale')
        case 3:
            ax.set_xscale('log')
            ax.set_yscale('log')
            plt.title('ax.set_xscale & ax.set_yscale')
        case 4:
            plt.loglog()
            plt.title('plt.loglog')
        case 5:
            plt.ylim(0, 2000)
            plt.semilogx()
            plt.title('plt.semilogx')
        case 6:
            plt.semilogy()
            plt.title('plt.semilogy')

plt.show()

```

log scales



- There are many ways to make the x- and/or y-axis logarithmic with pyplot

Saving figures

matplotlib-savefig.py

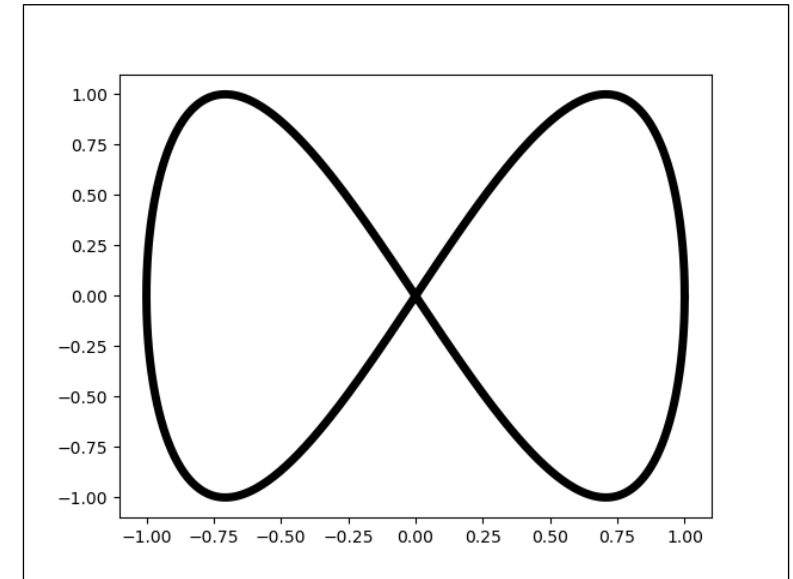
```
import matplotlib.pyplot as plt
from math import pi, sin, cos

n = 1000
points = [(cos(2 * pi * i / n),
           sin(4 * pi * i / n)) for i in range(n)]
x, y = zip(*points)
plt.plot(x, y, 'k-', linewidth=5)

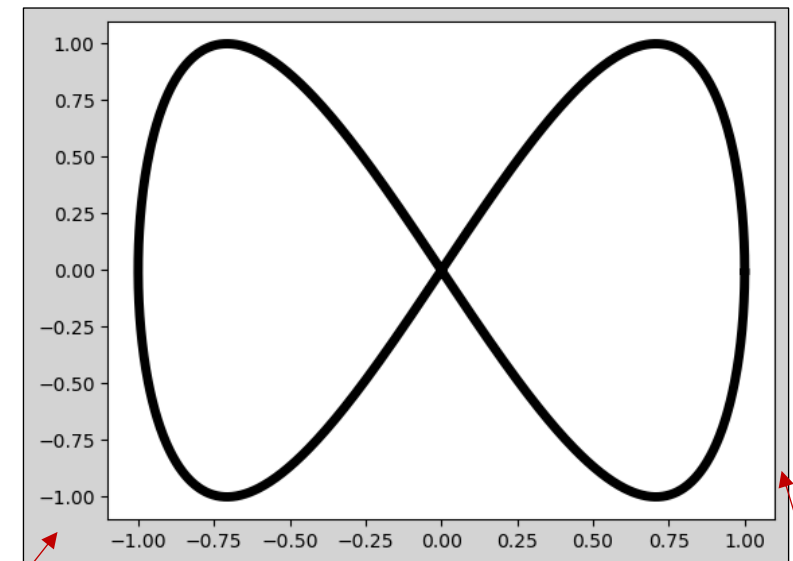
plt.savefig('butterfly.png') # save plot as PNG

plt.savefig('butterfly-grey.png',
            dpi=100,          # dots per inch
            bbox_inches='tight', # crop to bounding box
            pad_inches=0.1,    # space around figure
            facecolor='lightgrey', # background color
            format='png')      # optional if file extension

plt.savefig('butterfly.pdf') # save plot as PDF
plt.show()                  # interactive viewer
```



butterfly.png



butterfly-grey.png

facecolor

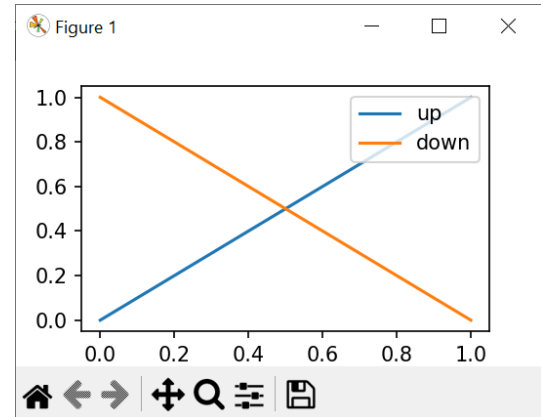
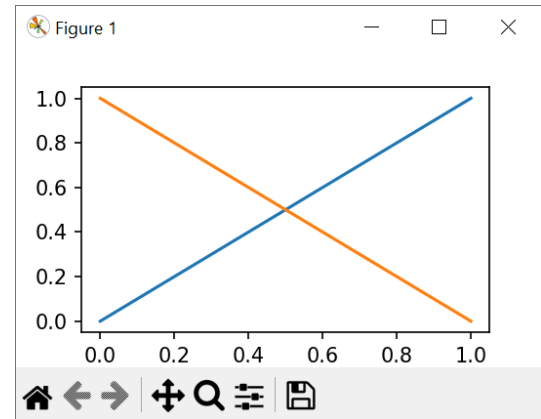
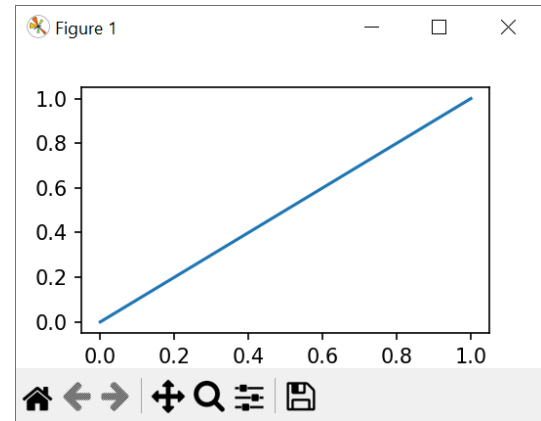
pad_inches

Interactive mode

Python shell

```
> import matplotlib.pyplot as plt
> plt.ion()                                     # Enable interactive mode
> plt.plot([0, 1], [0, 1], label='up')          # Shows plot immediately
> plt.plot([0, 1], [1, 0], label='down')        # Adds visible line
> plt.legend(loc='upper right')                 # Adds visible legend
> plt.ioff()                                    # Disable interactive mode
```

- Useful when developing plot from Python shell
- Automatically shows / updates plot



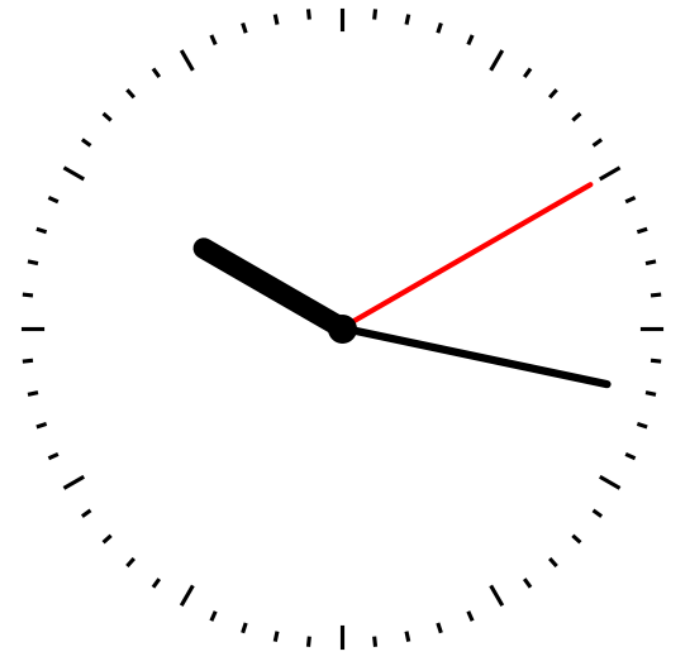
A crude animation

clock.py

```
import matplotlib.pyplot as plt
from math import pi, sin, cos
import datetime

def plot_clock(hour, minute, second):
    plt.axis('off')                # hide x and y axes
    plt.gca().set_aspect('equal')  # don't squeeze circle
    for i in range(60):           # show second marks
        angle = 2 * pi * i / 60
        x, y = cos(angle), sin(angle)
        start = 0.98 if i % 5 else .94 # every 5'th mark should be longer
        plt.plot([start * x, x], [start * y, y], c='black') # mark
    for angle, length, style in [
        (second / 60, 0.9, dict(c='red', lw=2)),
        (minute / 60, 0.85, dict(c='black', lw=3)),
        ((hour + minute / 60) / 12, 0.5, dict(c='black', lw=8))
    ]:
        angle = 2 * pi * (0.25 - angle)
        x, y = length * cos(angle), length * sin(angle)
        plt.plot([0, x], [0, y], **style, solid_capstyle='round') # arm
    plt.plot(0, 0, 'o', ms=10, c='black') # center dot

while True:
    now = datetime.datetime.now() # local time
    plot_clock(now.hour, now.minute, now.second)
    plt.pause(1) # show figure and pause 1 second
    plt.clf()    # clear figure
```



```

import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from math import pi, cos, sin

n, tail_length = 200, 75
points = [] # tail_length recent points

def point(i):
    t = 2 * pi * i / n
    return (cos(3 * t), sin(2 * t))

fig = plt.figure() # new figure
ax = plt.gca() # get current axes
ax.set_facecolor('black') # set background color
plt.xlim(-1.1, 1.1) # set x-axis range
plt.ylim(-1.1, 1.1) # set y-axis range
plt.xticks([]) # remove x-ticks & labels
plt.yticks([]) # remove y-ticks & labels
plt.title('Moving point') # plot title

x, y = point(0)
plt.plot(x, y, 'w.') # start point
plt.text(x - 0.025, y, 'start', color='w', # text label
         ha='right', va='center') # alignment
tail, = plt.plot([], [], 'w-', alpha=0.5) # init. tail
head, = plt.plot([], [], 'ro') # init. current point

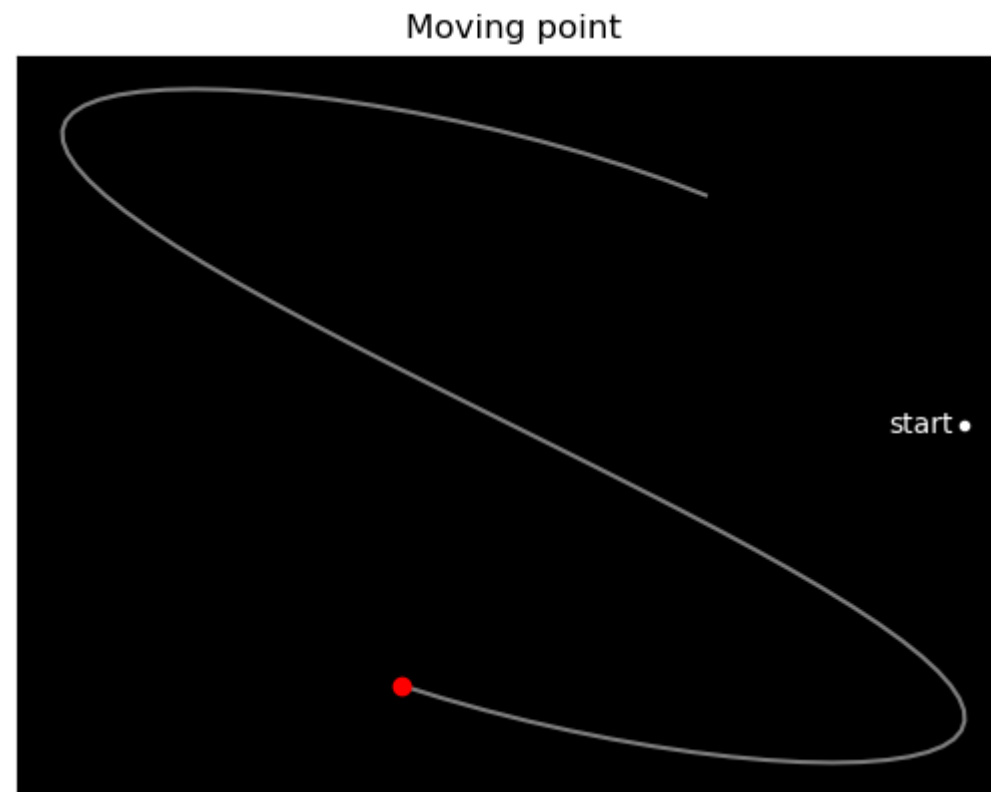
def move(frame): # frame = value from frames
    x, y = point(frame)
    points.append((x, y))
    del points[:-tail_length] # limit tail
    tail.set_data(*zip(*points)) # update tail points
    head.set_data([x], [y]) # update head point

animation = FuncAnimation(fig, # figure to animate
                          func=move, # function called for each frame
                          frames=range(n), # array like to iterate over
                          interval=25, # milliseconds between frames
                          repeat=True, # repeat frames when done
                          repeat_delay=0) # wait milliseconds before repeat

plt.show()

```

matplotlib.animation.FuncAnimation



- `plot` returns “Line2D” objects representing the plotted data
- “Line2D” objects can be updated using `set_data`
- To make an animation you need to repeatedly update the “line2D” objects
- `FuncAnimation` repeatedly calls `func` in regular intervals `interval`, each time with the next value from `frames` (if `frames` is None, then the frame values provided to `func` will be the infinite sequence 0,1,2,3,...)

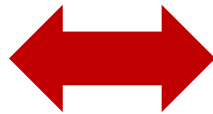


The Jupyter Notebook

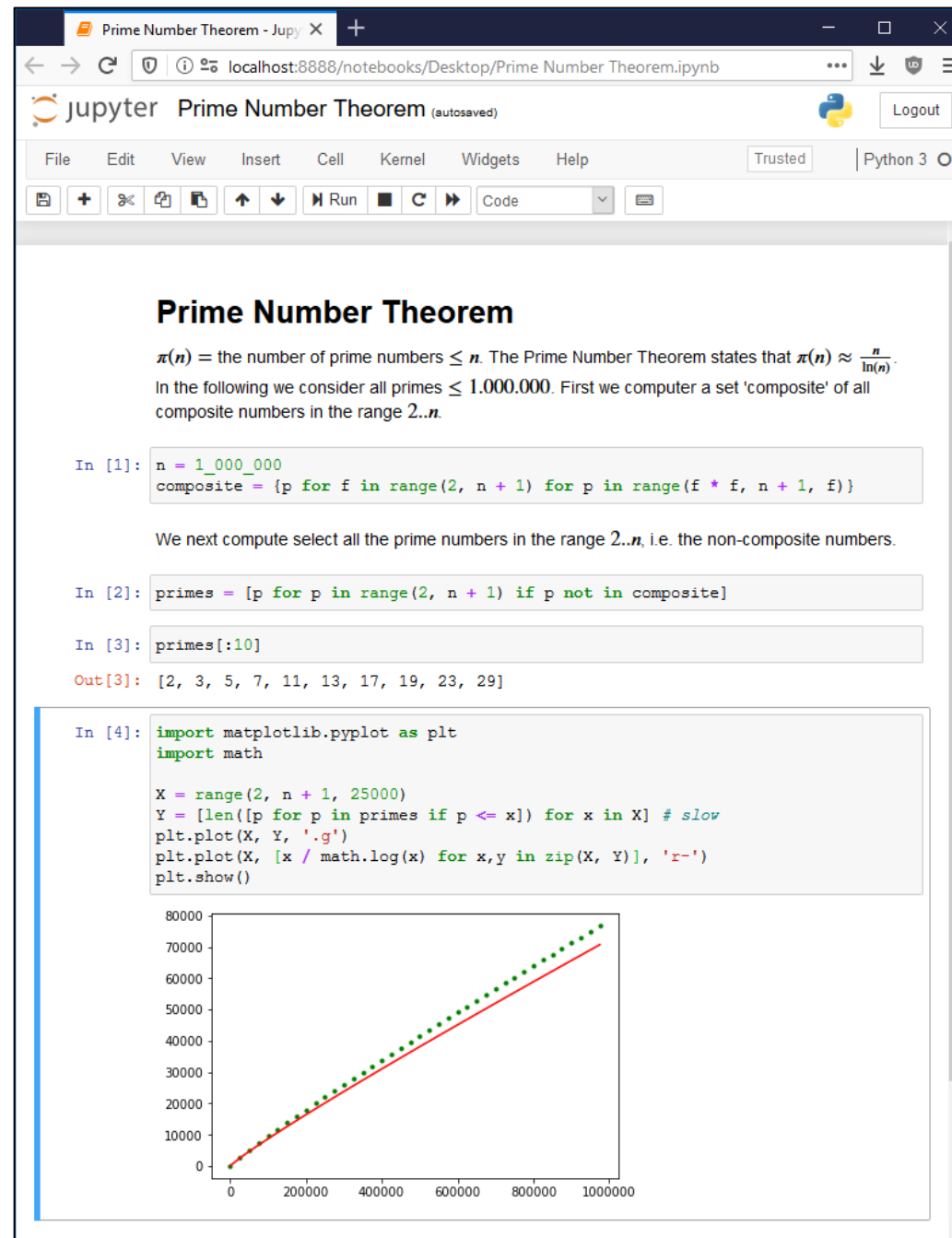
The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.



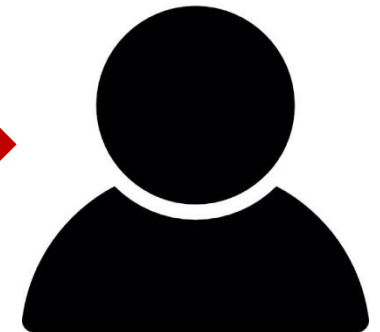
IP[y]:
IPython



Jupyter Server
(e.g. running on
local machine)



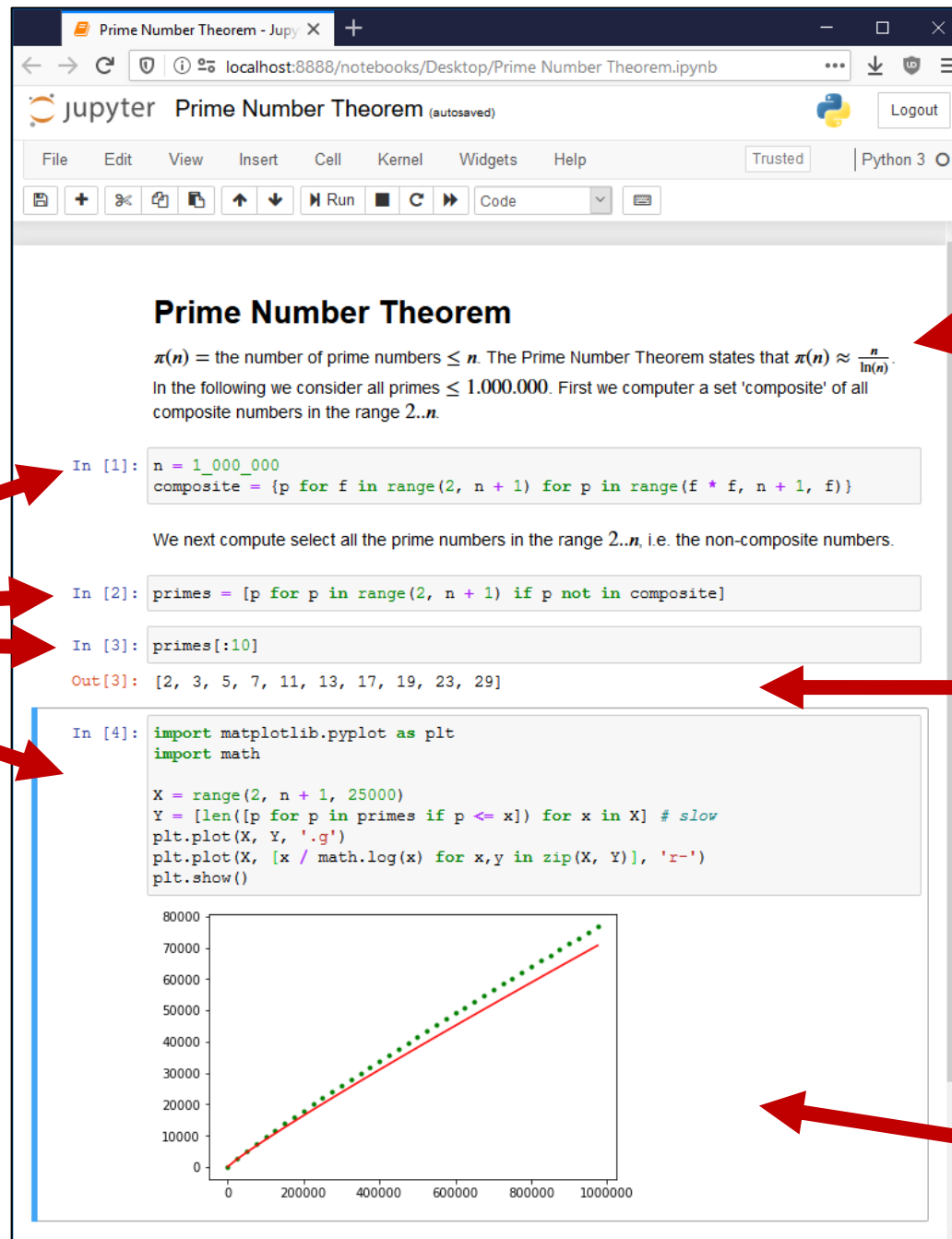
Web Browser



User

cells

python code



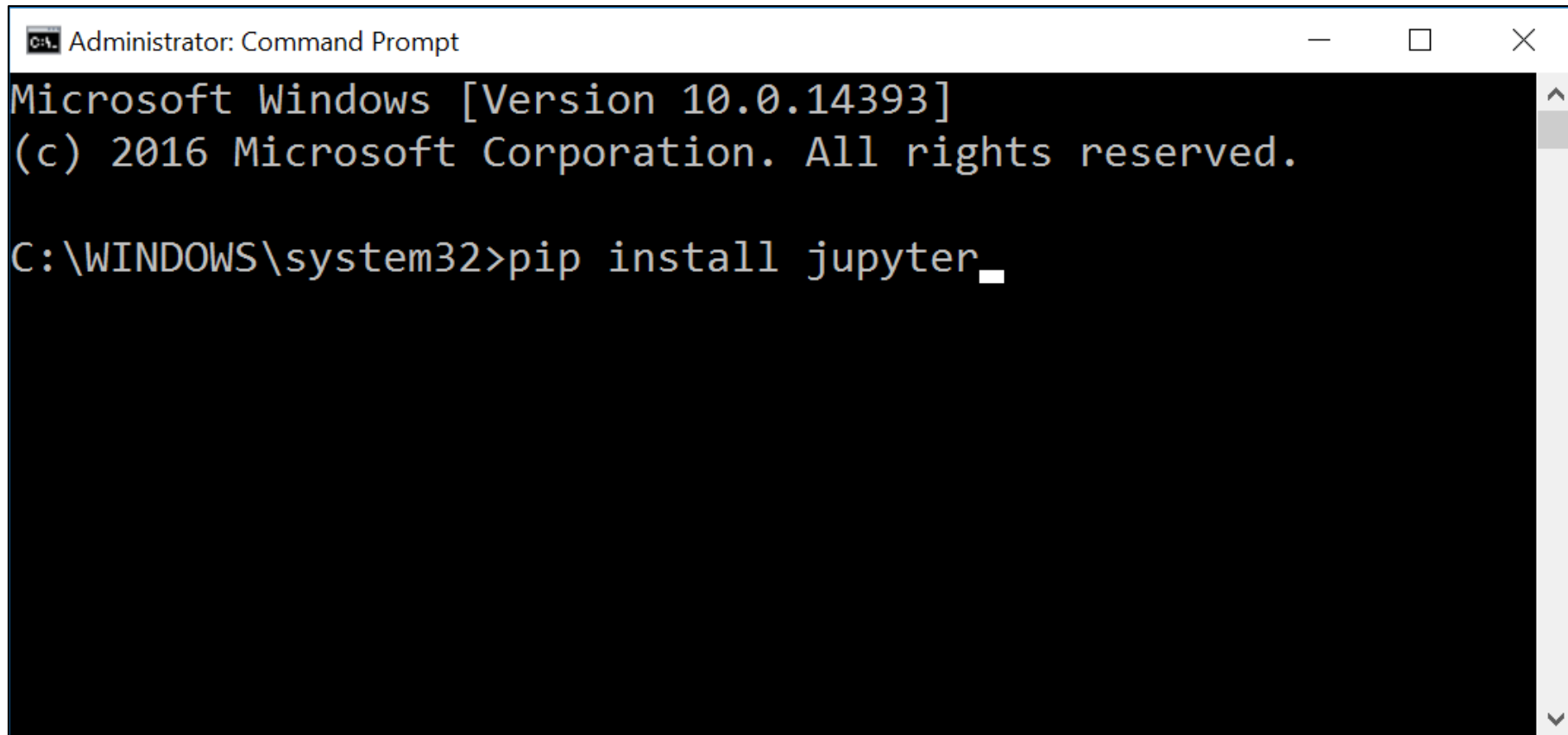
formatted text:
Markdown /
LaTeX / HTML /
...

python shell
output

matplotlib /
numpy / ...
output

Jupyter - installing

- Open a windows shell and run: `pip install jupyter`



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>pip install jupyter_
```

Jupyter – launching the jupyter server

- Open a windows shell and run: `jupyter notebook`



```
Command Prompt - jupyter notebook
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>jupyter notebook
[I 17:36:08.584 NotebookApp] Serving notebooks from local directory: C:\Users\au121
[I 17:36:08.584 NotebookApp] 0 active kernels
[I 17:36:08.584 NotebookApp] The Jupyter Notebook is running at:
[I 17:36:08.584 NotebookApp] http://localhost:8888/?token=18644cfeedab986bbdcc068f61c21b06b62548a090169463
[I 17:36:08.584 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 17:36:08.584 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://localhost:8888/?token=18644cfeedab986bbdcc068f61c21b06b62548a090169463
[I 17:36:09.154 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```

← url

- If this does not work, then try `python -m notebook`

Desktop/jupyter/

localhost:8888/tree/Desktop/jupyter

jupyter

Quit Logout

Files Running Clusters

Select items to perform actions on them.

Upload New

0 / Desktop / jupyter

..

Prime Number Theorem.ipynb

Python 3

Text File

Folder

Terminals Unavailable

create new notebook

Untitled - Jupyter Notebook

localhost:8888/notebooks/Desktop/jupyter/Untitled.ipynb?kernel_name=py

jupyter **Untitled** (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

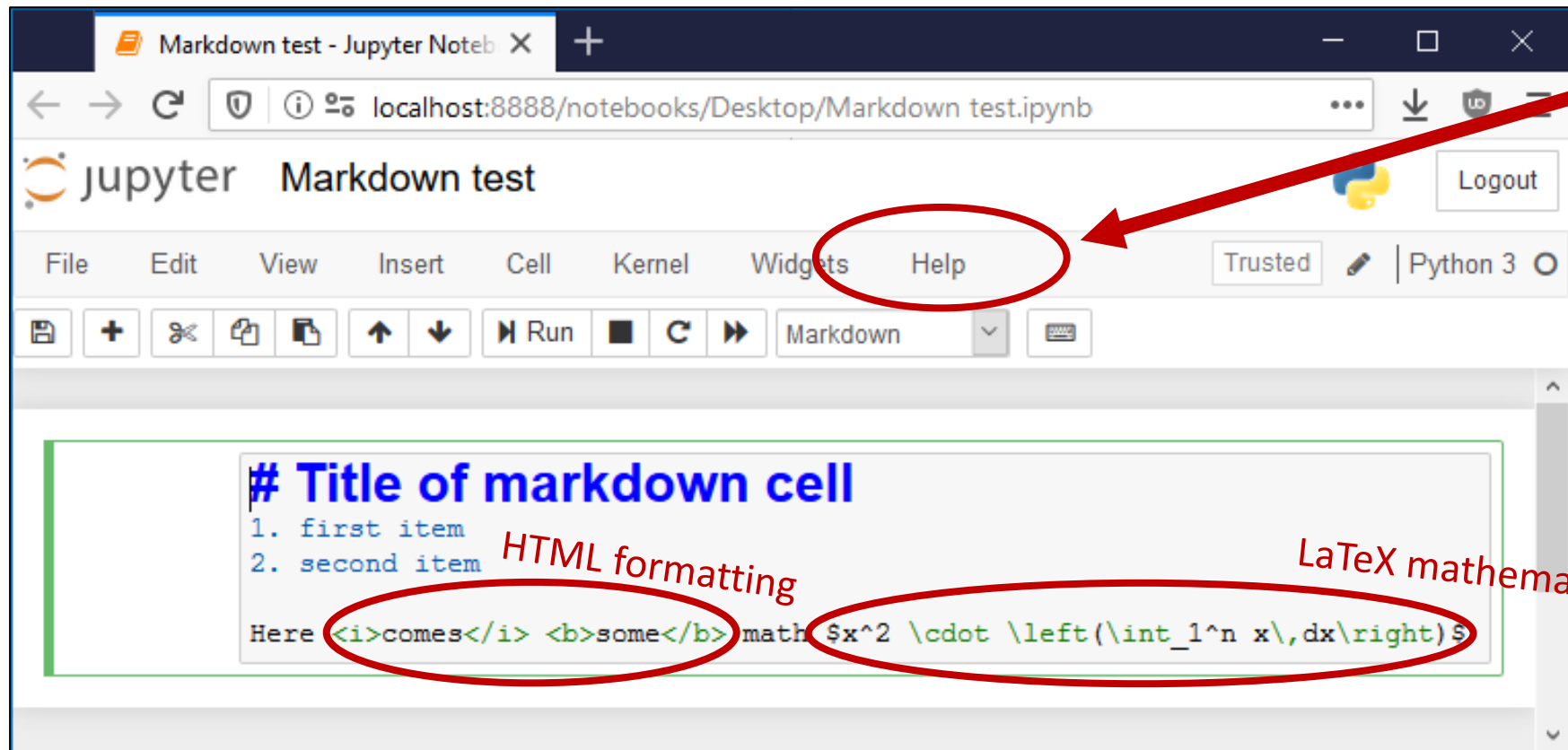
Code

In []: |

title - double click to change

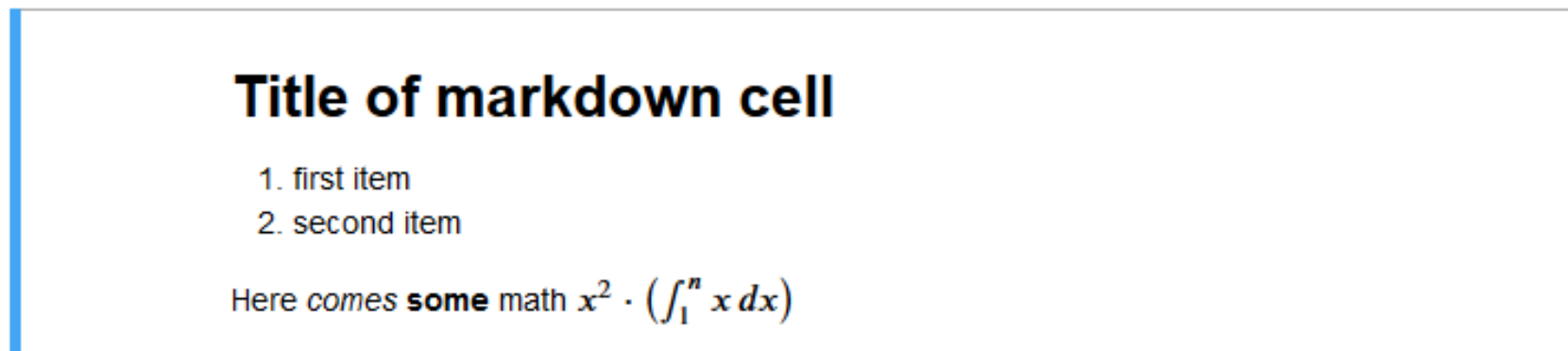
type of active cell

active cell



Try:
Help > User Interface Tour
Help > Markdown

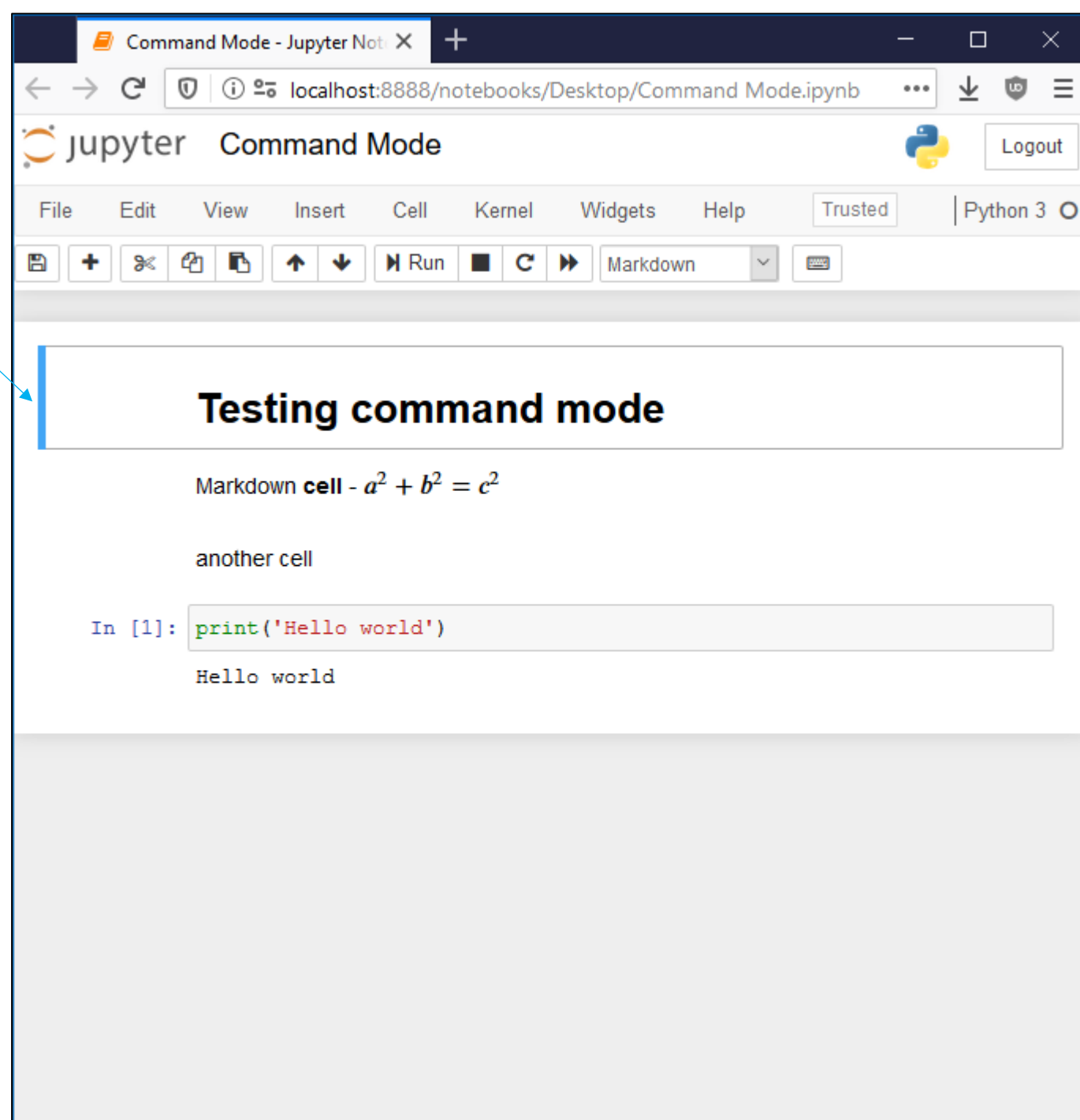
after pressing
Ctrl + Enter (evaluates)
Alt + Enter (evaluates + new cell)



Command Mode

- Used to navigate between cells
- Current cell is marked with blue bar
- Keyboard shortcuts

ctrl-shift-h	show keyboard shortcuts
enter	enter Edit Mode on current cell
escape	leave Edit Mode
shift-enter	run cell + select below
ctrl-enter	run selected cells
alt-enter	run cell and insert below
Y M R	change cell type (code, markdown, raw text)
1 2 3 4 5 6	change heading level
ctrl-A	select all cells
down up	move to next/previous cell
space shift-space	scroll down/up
shift-up shift-down	extend selected cells
A B	insert cell above/below
X C V shift-V Z DD	cut, copy, paste below/above, undo, delete cells
shift-L	toggle line numbers in cells
shift-M	merge selected cells (or with cell below)



The screenshot shows the Jupyter Notebook interface in Command Mode. The browser address bar indicates the notebook is located at `localhost:8888/notebooks/Desktop/Command Mode.ipynb`. The Jupyter logo and the text "Command Mode" are visible in the top left. A "Logout" button is in the top right. The menu bar includes "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". The "Trusted" status is shown on the right. The toolbar contains icons for saving, creating new cells, deleting cells, moving cells up and down, running cells, and a dropdown menu currently set to "Markdown". The main content area shows a notebook with two cells. The first cell is a Markdown cell with the heading "Testing command mode" and the text "Markdown cell - $a^2 + b^2 = c^2$ ". The second cell is a Code cell containing the Python code `print('Hello world')` and its output "Hello world". A blue vertical bar on the left side of the first cell indicates it is the current cell. A blue arrow points from the text "Current cell is marked with blue bar" in the list to this blue bar.

Edit Mode

- Used to edit current cell
- Current cell is marked with green bar
- Keyboard shortcuts

esc	enter Command Mode
shift-enter	run cell + select below
ctrl-enter	run selected cells
alt-enter	run cell and insert below
ctrl-shift--	split cell at cursor
tab	indent or code completion
shift-tab	show docstring
ctrl-a -x -c -v -z -y	select all, cut, copy, paste, undo, redo
ctrl-d	delete line

The screenshot shows the Jupyter Notebook interface in Edit Mode. The browser address bar indicates the URL is localhost:8888/notebooks/Desktop/Edit Mode.ipynb. The Jupyter logo and 'Edit Mode' text are visible. A menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. A toolbar contains icons for saving, adding, deleting, and running cells, along with a dropdown menu currently set to 'Code'. The notebook content includes a title 'Testing edit mode' and a text block 'Here we compute 7 · 6, 2⁸ and 'Hello world''. Below this are three code cells. The first cell contains `print(7 * 6)` and has output `42`. The second cell contains `2 ** 8` and has output `Out[2]: 256`. The third cell, which is currently selected and highlighted with a green border, contains `print('Hello world')`. A tooltip is displayed over this cell, showing the docstring for the `print` function: `Docstring: print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)`. A green arrow points from the 'green bar' mentioned in the text to the green border of the selected cell. A red arrow points from the 'show docstring' shortcut in the table to the docstring tooltip.

Evaluating cells

- To evaluate cell
ctrl-enter, alt-enter, shift-enter
- Output from program shown below cell
- Result of last evaluated line
- Order of code cells evaluated
Note "x ** 2" computed after "x = 4"
- [*] are cells being evaluated / waiting
- [] not yet evaluated
- Recompute all cells top-down
▶▶ or Kernel > Restart & Run all

The screenshot shows a Jupyter Notebook window titled "Evaluation - Jupyter Notebook". The browser address bar shows "localhost:8888/notebooks/Desktop/Evaluation.ipynb". The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, adding, deleting, and running cells. The notebook contains several code cells:

- In [1]: `print(42)` with output `42`
- In [2]: `x = 3`
- In [5]: `x ** 2` with output `Out[5]: 16`
- In [4]: `x = 4`
- In [*]: `while True: pass` (cell is being evaluated)
- In []: `print('Hello world')` (cell is not yet evaluated)

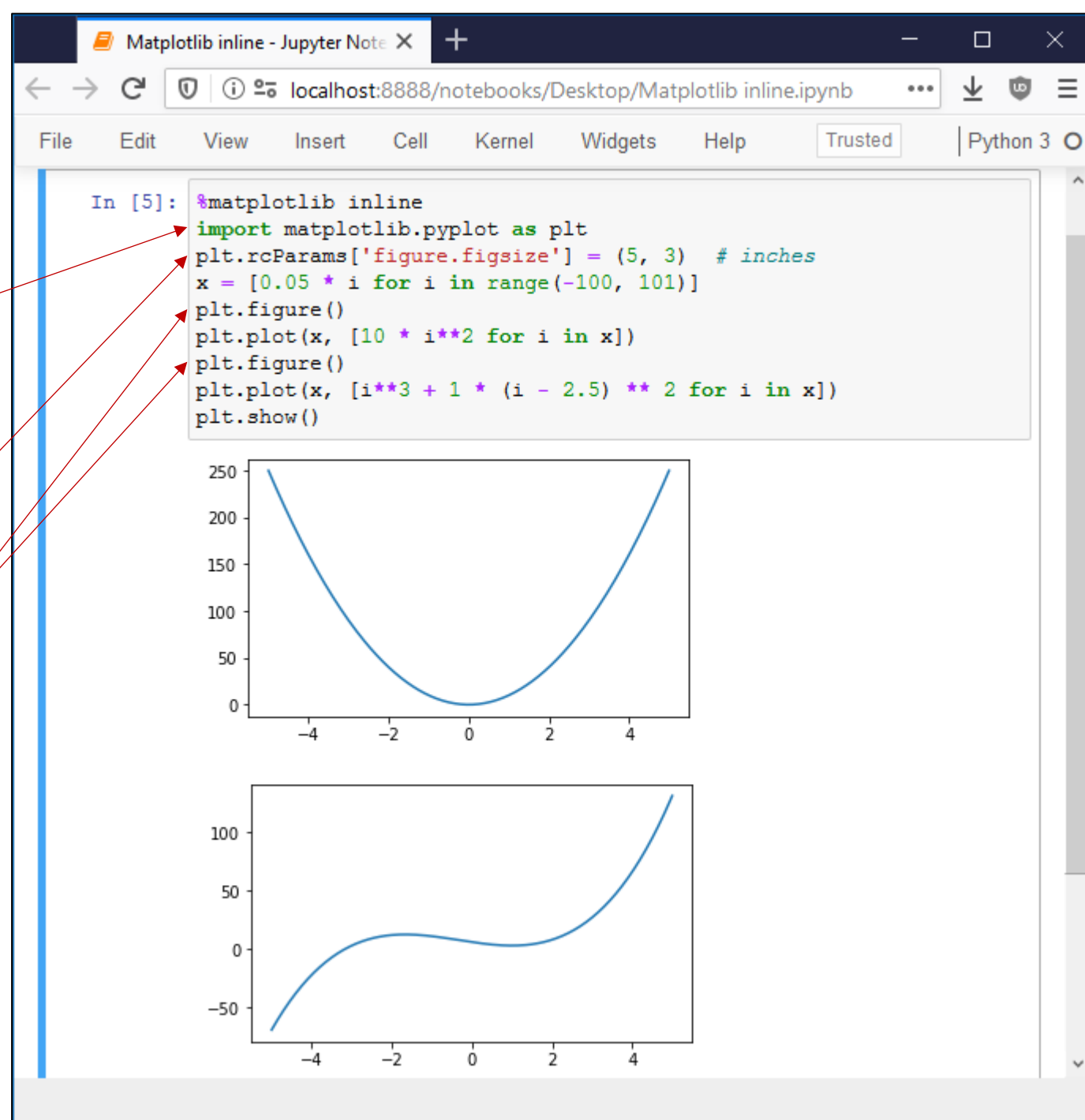
Magic lines

- Jupyter code cells support *magic commands* (actually it is IPython)
- % is a *line magic*
- %% is a *cell magic*

%lsmagic	list magic commands
%quickref	quick reference sheet to IPython
%pwd	print working directory (current folder)
%cd <i>directory</i>	change directory (absolut or relative)
%ls	list content of current directory
%pip or %conda	run pip or conda from jupyter
%load <i>script</i>	insert external script into cell
%run <i>program</i>	run external program and show output
%automagic	toggle if %-prefix is required
%matplotlib inline	no zoom & resize, allows multiple plots
%matplotlib widget	a single plot can be zoomed & resized
%%writefile <i>file</i>	write content of cell to a file
%%time	measure time for cell execution
%timeit <i>expression</i>	time for simple expression

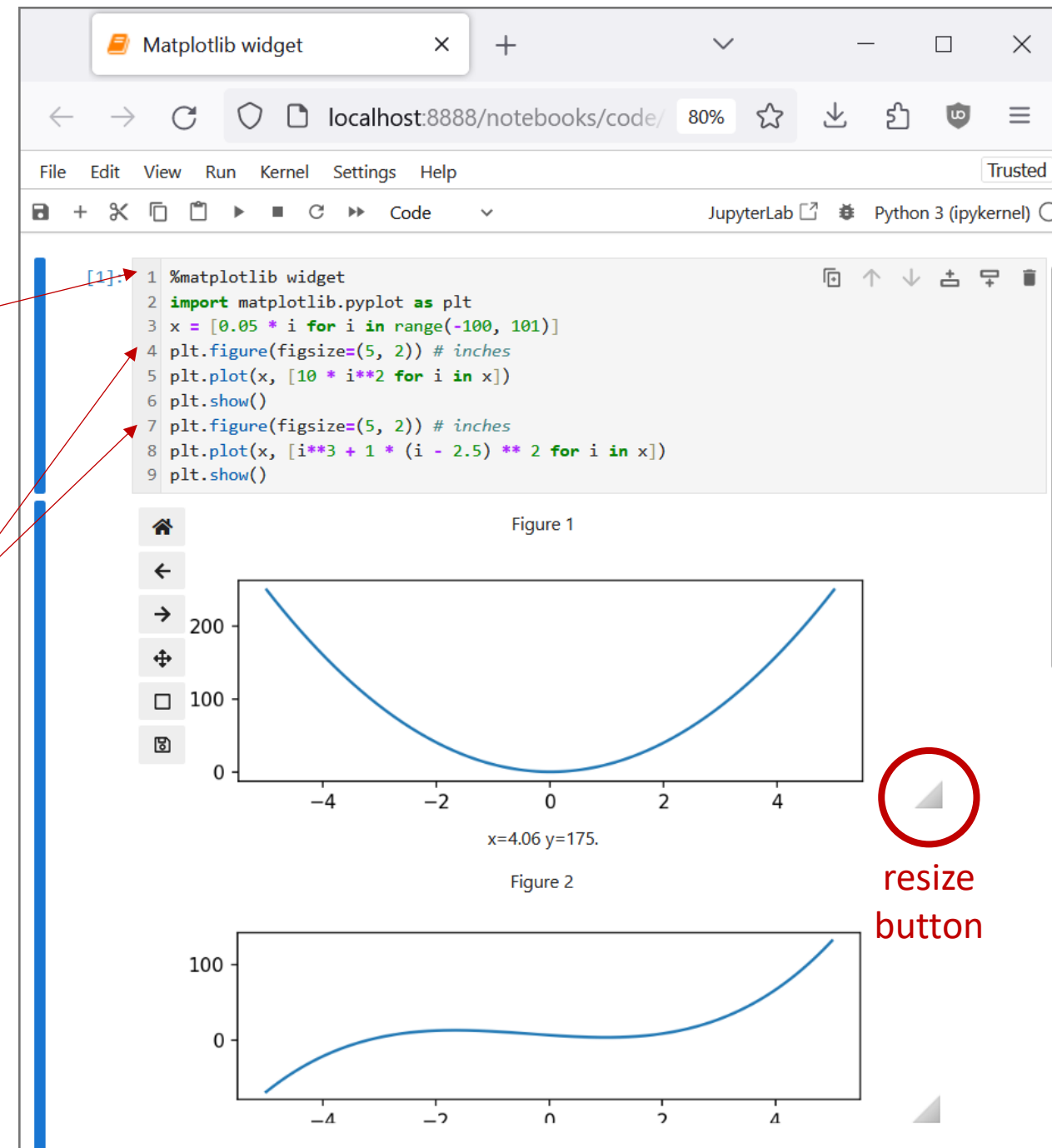
Jupyter and matplotlib

- `%matplotlib inline`
pyplot figures are shown *without* interactive zoom and pan (default)
- Consider changing default figure size
`plt.rcParams['figure.figsize']`
- Start each figure with `plt.figure()`
- Final call to `show` can be omitted



Jupyter and matplotlib

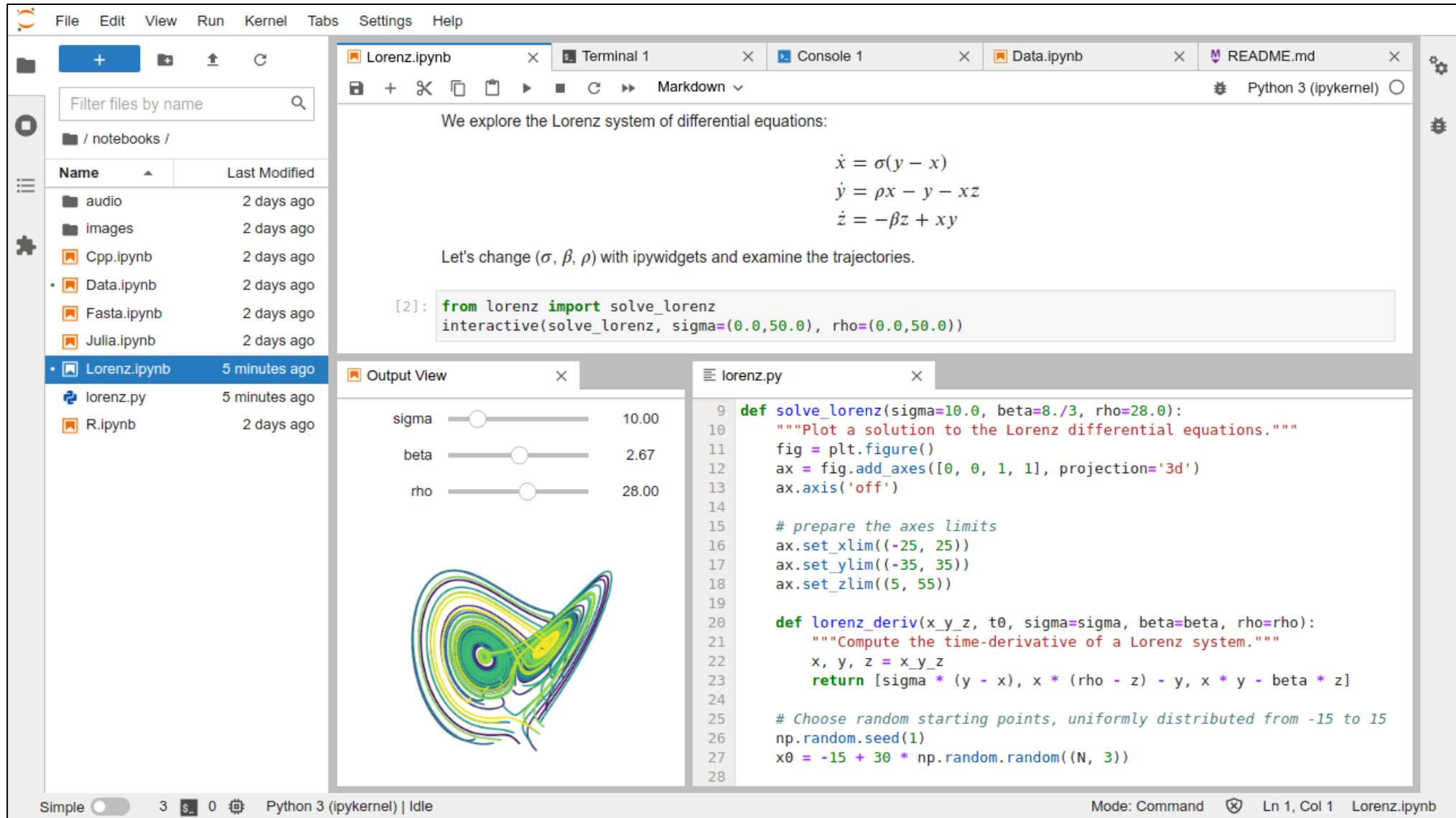
- `%matplotlib widget`
pyplot figures are shown *with* interactive zoom and pan
`pip install ipympl`
- Start each figure with `plt.figure`
(also allows setting figure size)
- Final call to `show` can be omitted





- Widespread tool used for data science applications
- Documentation, code for data analysis, and resulting visualizations are stored in one common format
- Easy to update visualizations
- Works with about 100 different programming languages (not only Python 3), many special features,
- Easy to share data analysis
- IDEs with Notebook support: VS Code, Spyder, PyCharm
- Online Jupyter Notebook with no setup: colab.google
- *Many online tutorials and examples are available*
https://www.youtube.com/results?search_query=jupyter+python

JupyterLab: A Next-Generation Notebook Interface



The screenshot displays the JupyterLab interface with the following components:

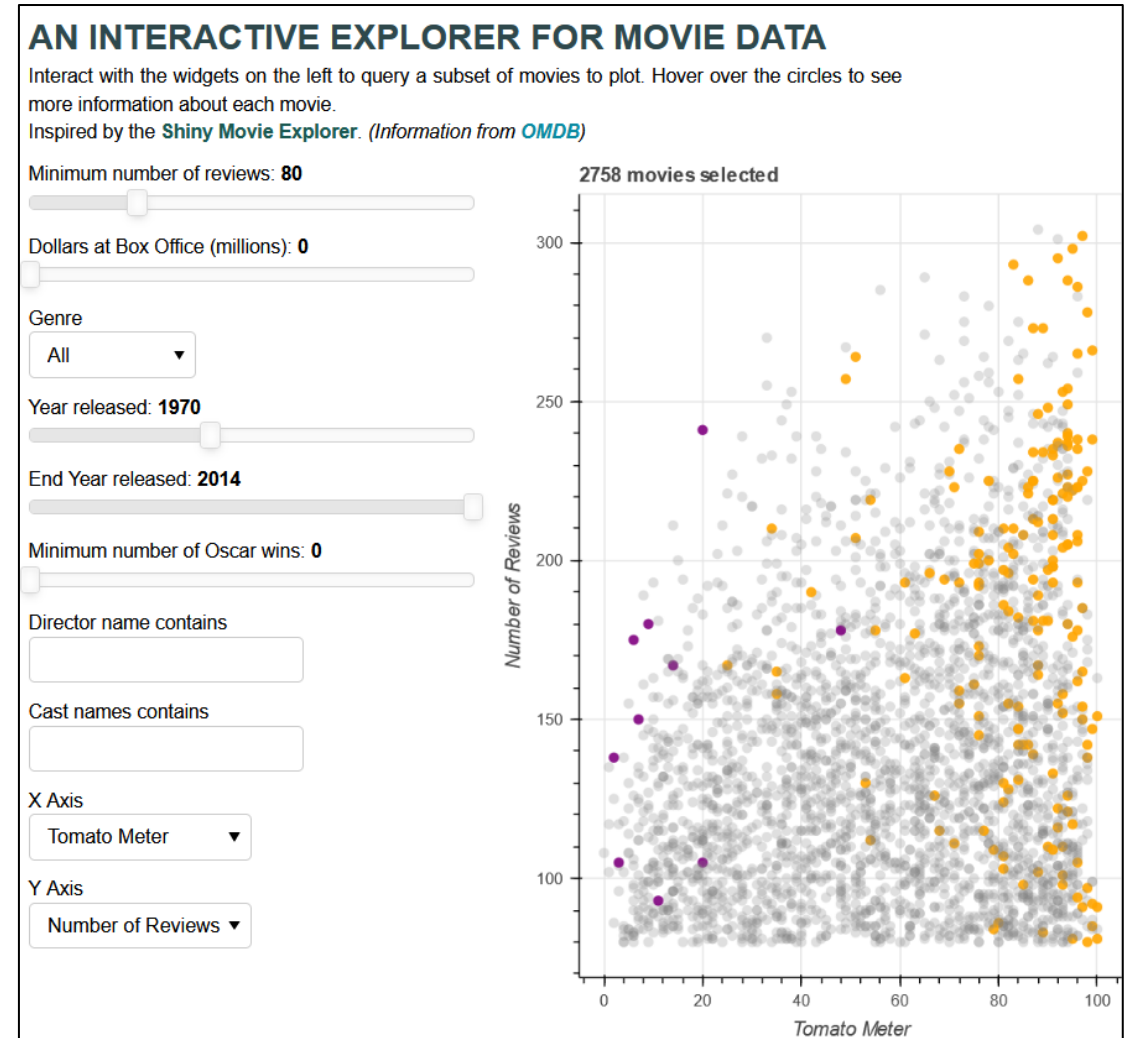
- File Browser (Left):** Shows a file tree with folders like 'audio' and 'images', and notebooks like 'Cpp.ipynb', 'Data.ipynb', 'Fasta.ipynb', 'Julia.ipynb', 'Lorenz.ipynb' (selected), 'lorenz.py', and 'R.ipynb'.
- Notebook (Top):** The 'Lorenz.ipynb' notebook is open, showing a text cell with the Lorenz system equations and a code cell with the following code:

```
[2]: from lorenz import solve_lorenz
      interactive(solve_lorenz, sigma=(0.0,50.0), rho=(0.0,50.0))
```
- Output View (Bottom Left):** Displays interactive sliders for parameters: sigma (0.0 to 10.00), beta (0.0 to 2.67), and rho (0.0 to 28.00). Below the sliders is a 3D plot of the Lorenz attractor.
- Code Editor (Bottom Right):** Shows the source code for 'lorenz.py' with the following code:

```
9 def solve_lorenz(sigma=10.0, beta=8./3, rho=28.0):
10     """Plot a solution to the Lorenz differential equations."""
11     fig = plt.figure()
12     ax = fig.add_axes([0, 0, 1, 1], projection='3d')
13     ax.axis('off')
14
15     # prepare the axes limits
16     ax.set_xlim((-25, 25))
17     ax.set_ylim((-35, 35))
18     ax.set_zlim((5, 55))
19
20     def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
21         """Compute the time-derivative of a Lorenz system."""
22         x, y, z = x_y_z
23         return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]
24
25     # Choose random starting points, uniformly distributed from -15 to 15
26     np.random.seed(1)
27     x0 = -15 + 30 * np.random.random((N, 3))
28
```
- Terminal and Console (Top):** The 'Terminal 1' and 'Console 1' tabs are open but empty.
- Bottom Bar:** Shows the current mode as 'Command', the kernel as 'Python 3 (ipykernel)', and the file name 'Lorenz.ipynb'.

Visualization libraries

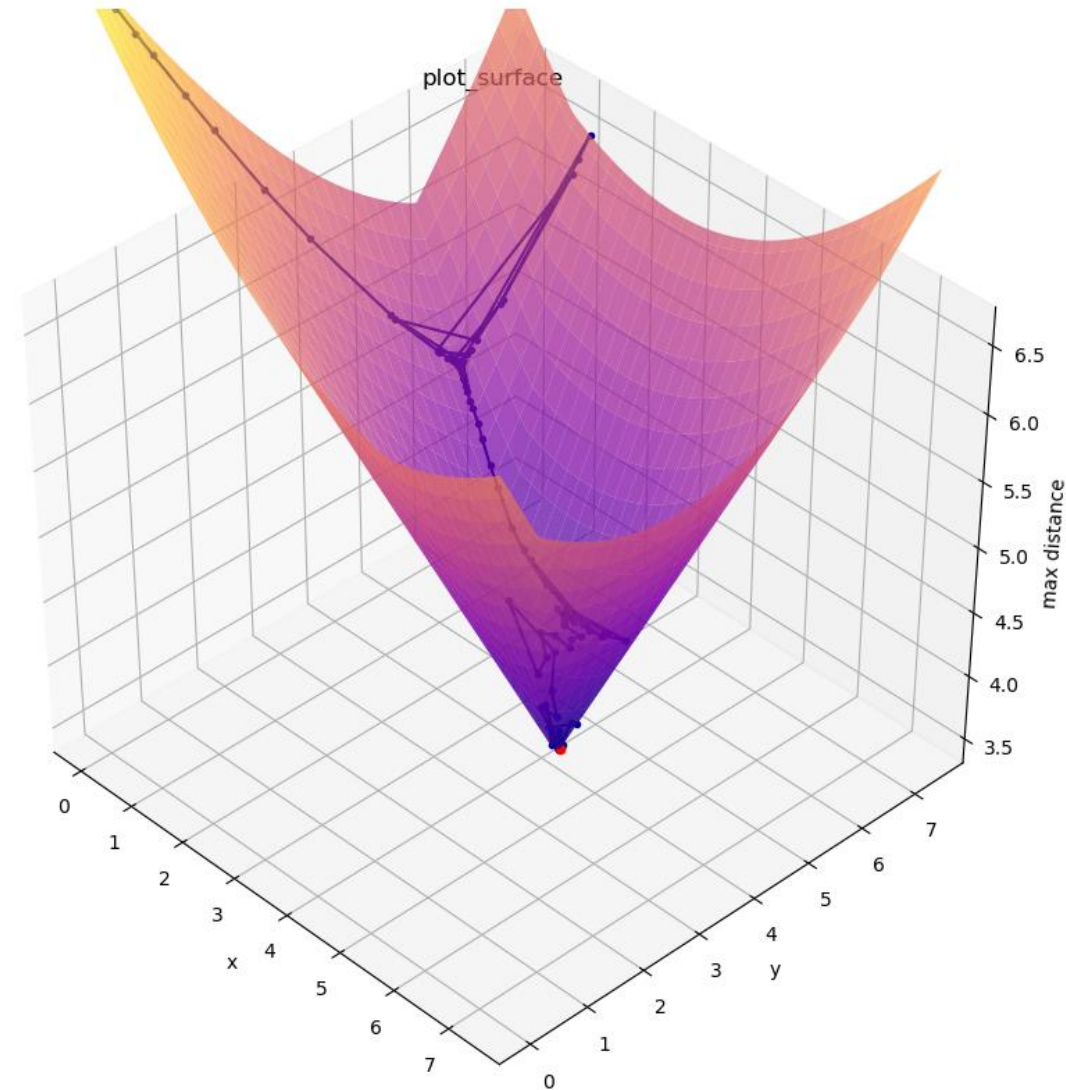
- Altair - altair-viz.github.io
- Bokeh - bokeh.org
- Plotly - plotly.com/python
- Seaborn - seaborn.pydata.org



demo.bokeh.org/movies

scipy.optimize.minimize

- Find point **p** minimizing function **f**
- Supports 13 algorithms – but no guarantee that result is correct
- Knowledge about optimization will help you know what optimization algorithm to select and what parameters to provide for better results
-  **WARNING** 
Many solvers return the wrong value when used as a black box



minimize.py

```
from math import sin
import matplotlib.pyplot as plt
from scipy.optimize import minimize

trace = [] # remember calls to f

def f(x):
    value = x[0]**2 + 10*sin(x[0])
    trace.append((x[0], value))
    return value

X = [-8 + 18 * i / 9999 for i in range(1000)]
Y = [f([x]) for x in X]

plt.style.use('dark_background')
plt.plot(X, Y, 'w-')
for start, color in [(8, 'red'), (-6, 'yellow')]:
    trace = []
    solution = minimize(f, [start], method='nelder-mead')

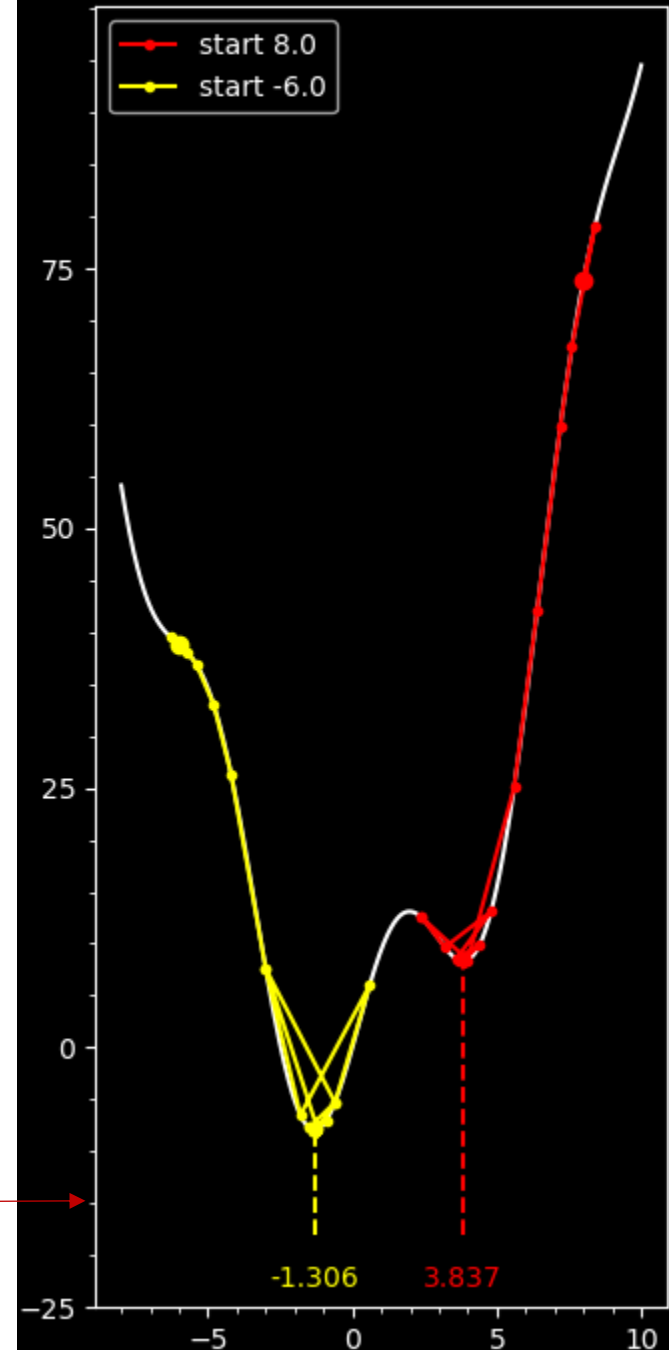
    x, y = solution.x[0], solution.fun
    plt.plot(*zip(*trace), '-.', c=color, label=f'start {start:.1f}') # trace
    plt.plot(*trace[0], 'o', c=color) # first trace point
    plt.text(x, -23, f'{x:.3f}', c=color, ha='center') # show minimum x
    plt.plot([x, x], [-18, y], '--', c=color) # dash to minimum

plt.xticks(range(-5, 15, 5))
plt.yticks(range(-25, 100, 25))
plt.minorticks_on()
plt.legend()
plt.show()
```

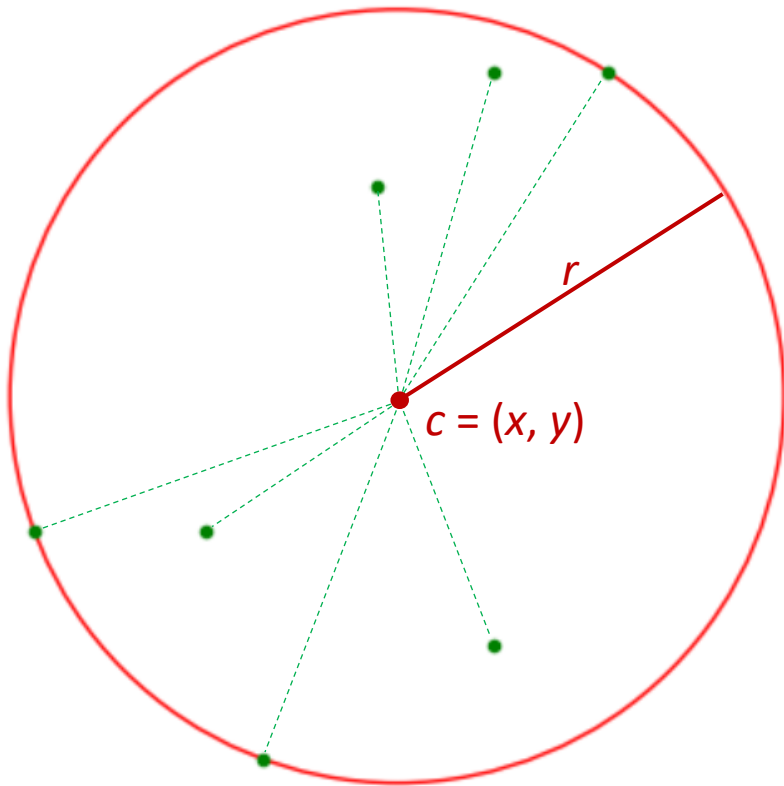
Python shell

```
> print(solution)
final_simplex: (array([[ -1.3064209 ],
 [-1.30649414]]), array([-7.94582337, -7.94582336]))
fun: -7.94582337348758
message: 'Optimization terminated successfully.'
nfev: 38
nit: 19
status: 0
success: True
x: array([ -1.3064209])
```

`minimize` tries to find a local minimum for f by repeatedly evaluating f for different x values



Example: Minimum enclosing circle



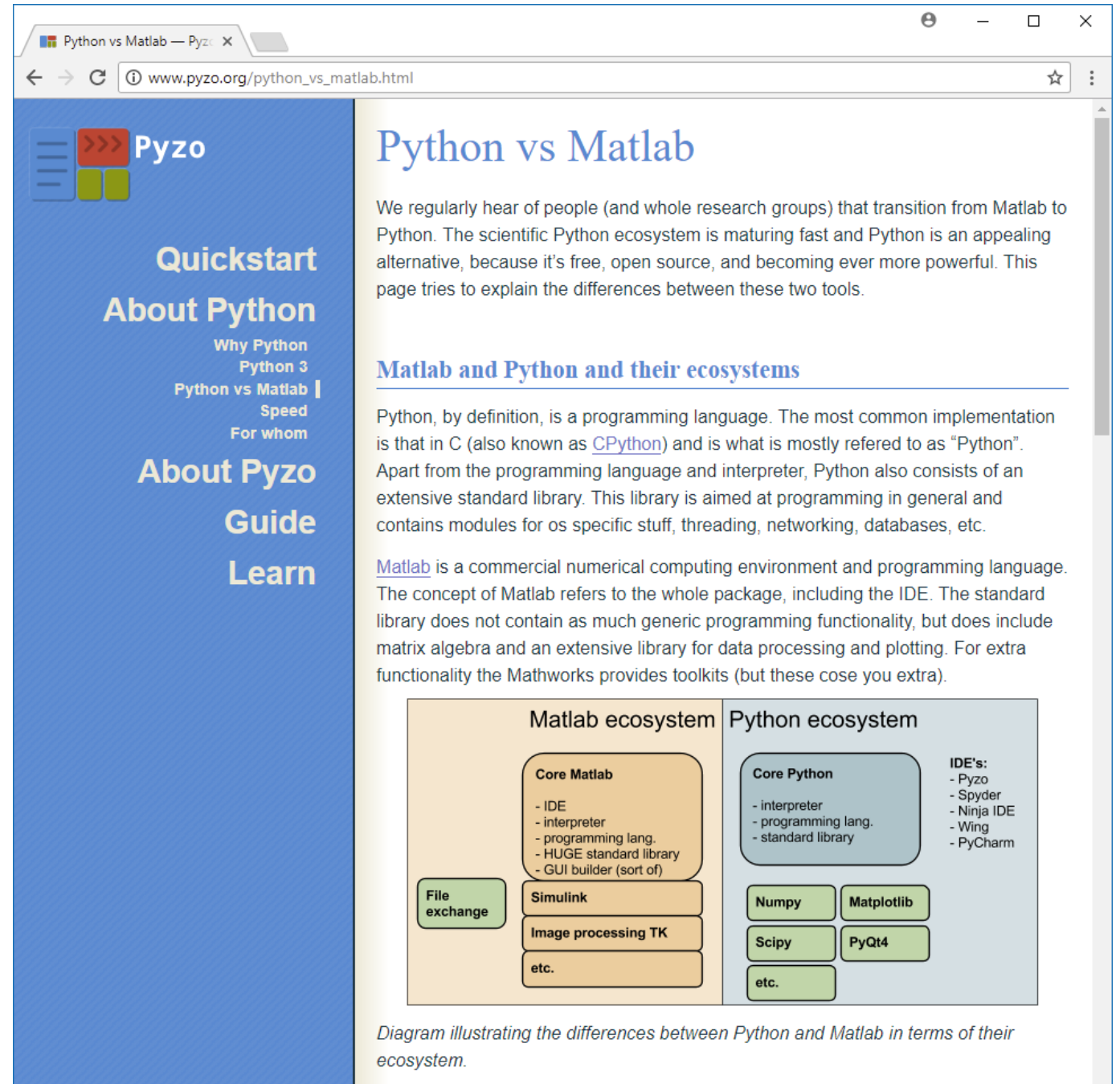
- Find c such that $r = \max_p |p - c|$ is **minimized**
- A solution is characterized by either
 - 1) three points on circle, where the triangle contains the circle center
 - 2) two opposite points on diagonal
- Try a standard numeric minimization solver
-  Computation involves **max** and $\sqrt{x}$, which can be hard for numeric optimization solvers

Python/scipy vs MATLAB

Some basic differences

- “**end**” closes a MATLAB block
- “**;**” at end of command avoids command output
- $a(i)$ instead $a[i]$
- 1st element of a list $a(1)$
- $a(i:j)$ includes both $a(i)$ and $a(j)$

like R, Mathematica, Julia, AWK, Smalltalk, ...



The screenshot shows a web browser window with the URL www.pyzo.org/python_vs_matlab.html. The page is titled "Python vs Matlab" and features a sidebar with navigation links: "Quickstart", "About Python" (with sub-links "Why Python", "Python 3", "Python vs Matlab", "Speed", "For whom"), "About Pyzo", "Guide", and "Learn". The main content area discusses the differences between the two ecosystems. It states that Python is a programming language with a large standard library, while Matlab is a commercial environment. A diagram at the bottom compares the "Matlab ecosystem" and the "Python ecosystem".

Matlab ecosystem

- Core Matlab
 - IDE
 - interpreter
 - programming lang.
 - HUGE standard library
 - GUI builder (sort of)
- File exchange
- Simulink
- Image processing TK
- etc.

Python ecosystem

- Core Python
 - interpreter
 - programming lang.
 - standard library
- IDE's:
 - Pyzo
 - Spyder
 - Ninja IDE
 - Wing
 - PyCharm
- Numpy
- Matplotlib
- Scipy
- PyQt4
- etc.

Diagram illustrating the differences between Python and Matlab in terms of their ecosystem.

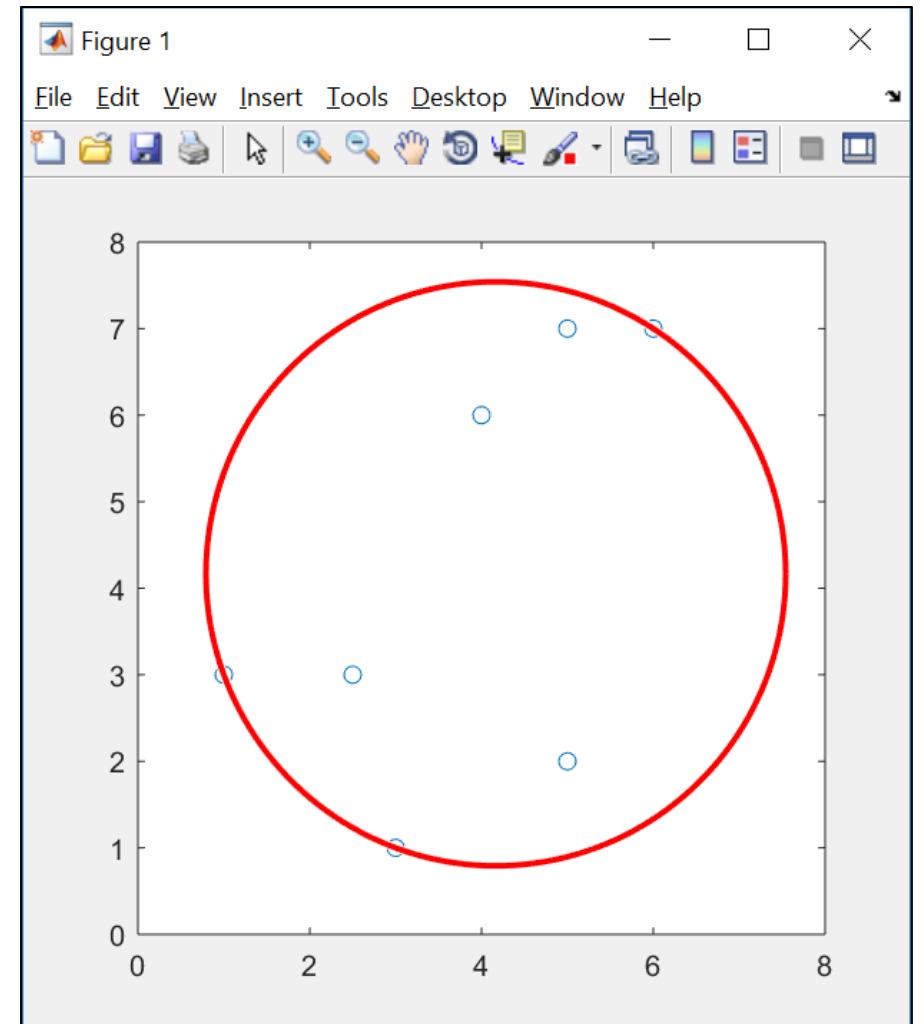
Minimum enclosing circle in MATLAB

enclosing_circle.m

```
% Minimum enclosing circle of a point set
% fminsearch uses the Nelder-Mead algorithm

global x y
x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0];
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0];
c = fminsearch(@(x) max_distance(x), [0,0]);
plot(x, y, "o");
viscircles(c, max_distance(c));

function dist = max_distance(p)
    global x y
    dist = 0.0;
    for i=1:length(x)
        dist = max(dist, pdist([p; x(i), y(i)],
                               'euclidean'));
    end
end
end
```



Minimum enclosing circle in MATLAB (trace)

enclosing_circle_trace.m

```
global x y trace_x trace_y

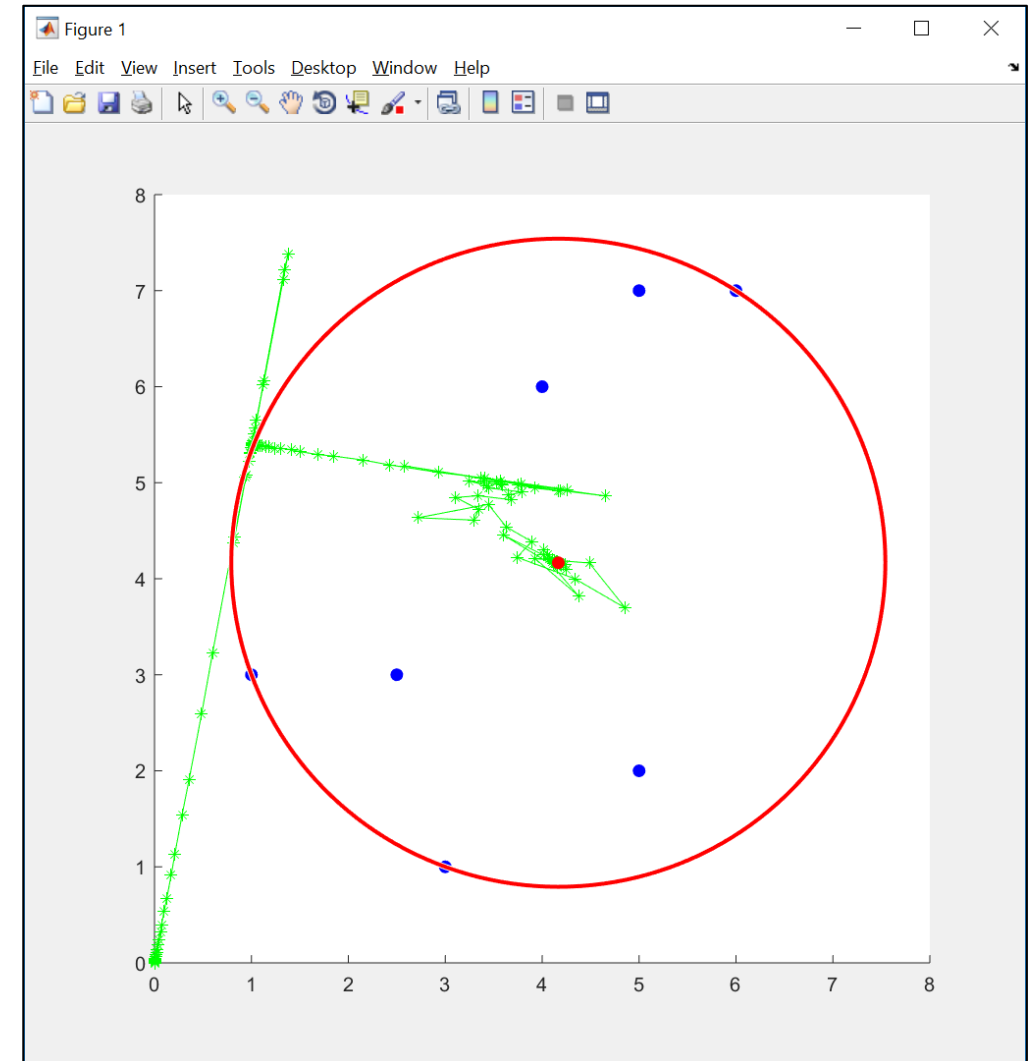
x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0];
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0];
trace_x = [];
trace_y = [];

c = fminsearch(@(x) max_distance(x), [0, 0]);

hold on
plot(x, y, "o", 'color', 'b', 'MarkerFaceColor', 'b');
plot(trace_x, trace_y, "*-", "color", "g");
plot(c(1), c(2), "o", 'color', 'r', 'MarkerFaceColor', 'r');
viscircles(c, max_distance(c), "color", "red");

function dist = max_distance(p)
    global x y trace_x trace_y
    trace_x = [trace_x, p(1)];
    trace_y = [trace_y, p(2)];

    dist = 0.0;
    for i=1:length(x)
        dist = max(dist, pdist([p; x(i), y(i)], 'euclidean' ));
    end
end
```



Minimum enclosing circle in Python

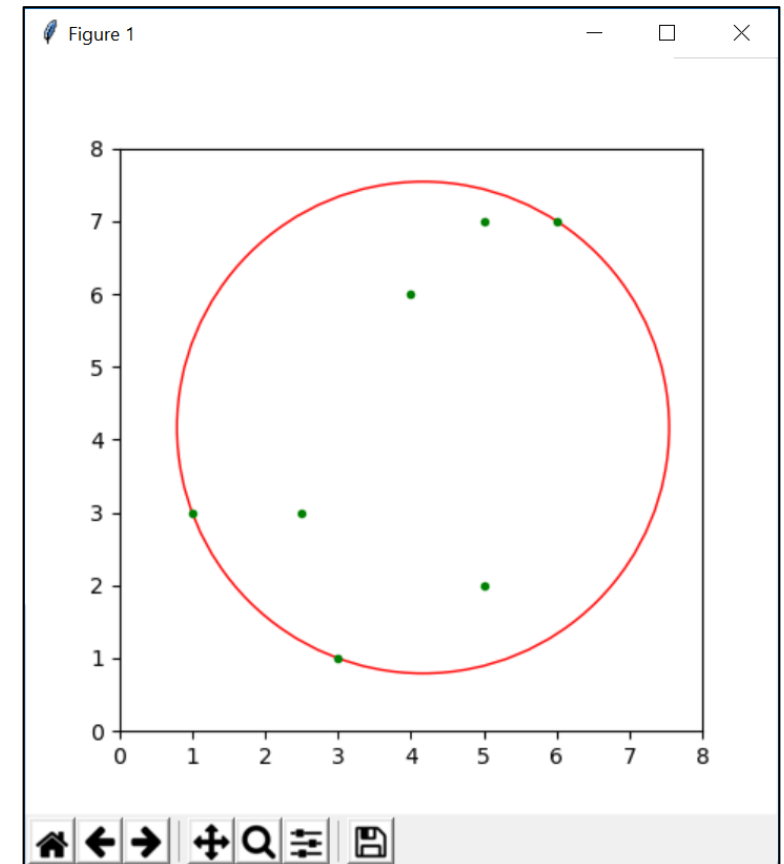
enclosing_circle.py

```
from scipy.optimize import minimize } import modules
import matplotlib.pyplot as plt

x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0]
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0]

def dist(p, q):
    return ((p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2) ** 0.5
def max_distance(c):
    return max([dist(p, c) for p in zip(x, y)])

c = minimize(max_distance, [0.0, 0.0], method='nelder-mead').x
ax = plt.gca()
ax.set_xlim((0, 8))
ax.set_ylim((0, 8))
ax.set_aspect('equal')
plt.plot(x, y, 'g.')
ax.add_artist(plt.Circle(c, max_distance(c),
                        color='r', fill=False))
plt.show()
```



Minimum enclosing circle in Python (trace)

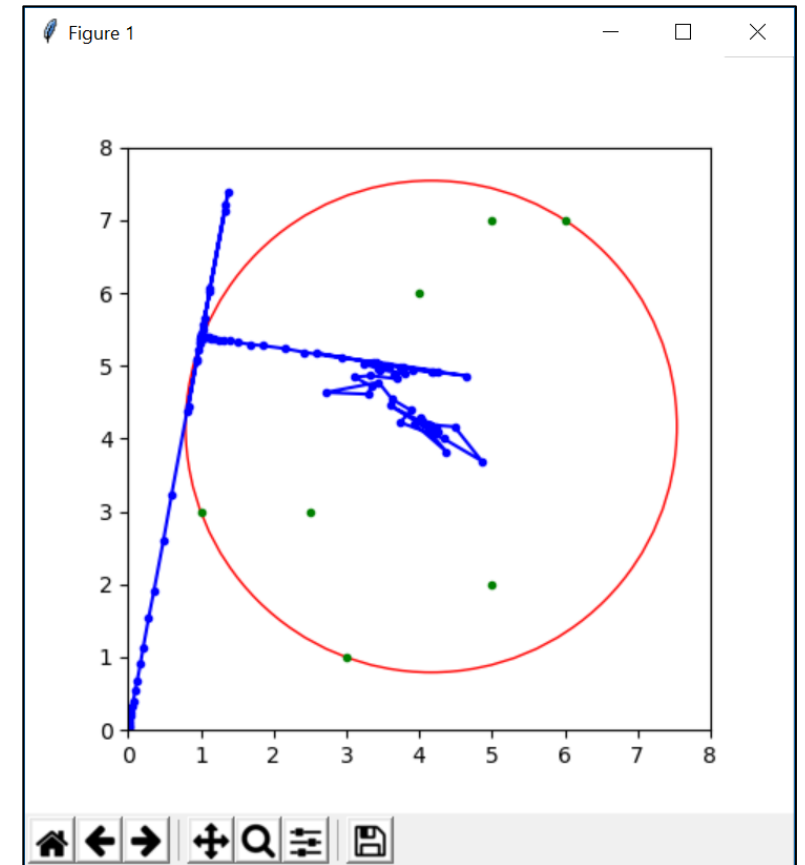
enclosing_circle_trace.py

```
from scipy.optimize import minimize
import matplotlib.pyplot as plt

x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0]
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0]
trace = []

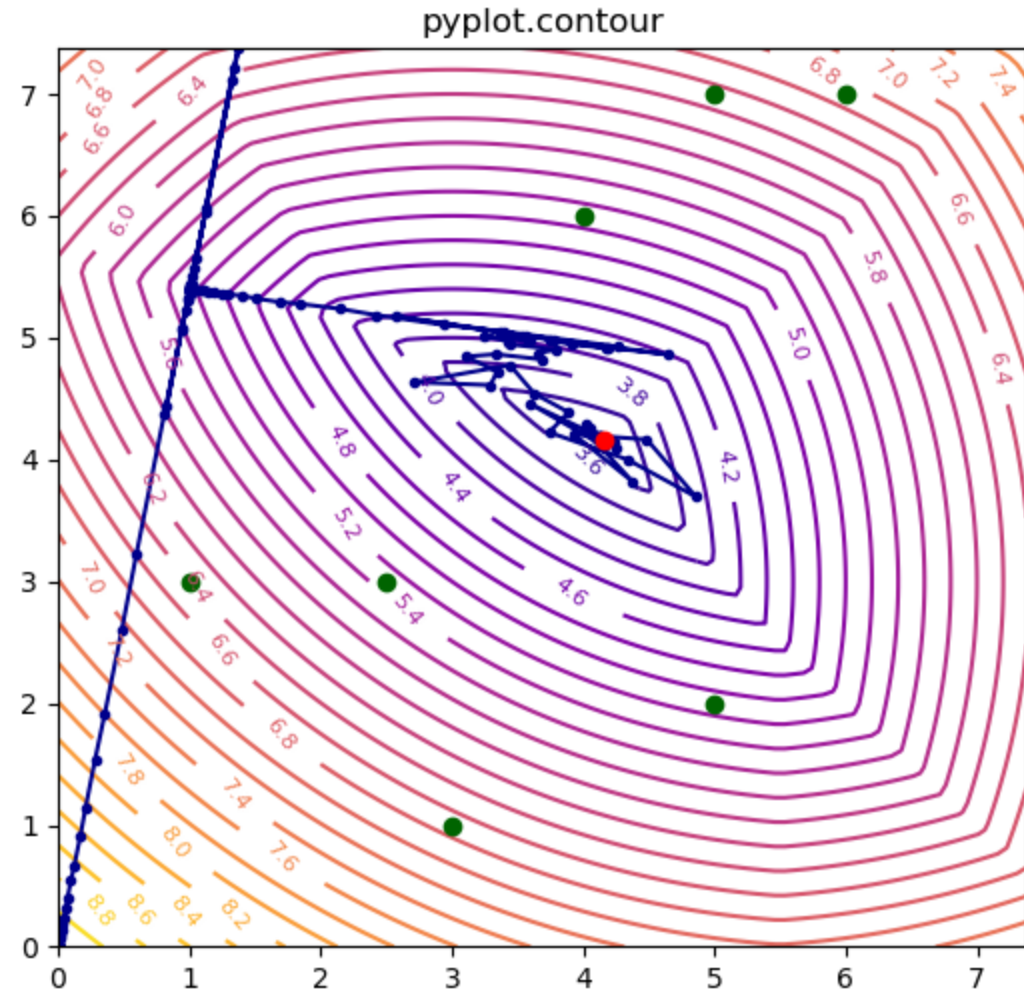
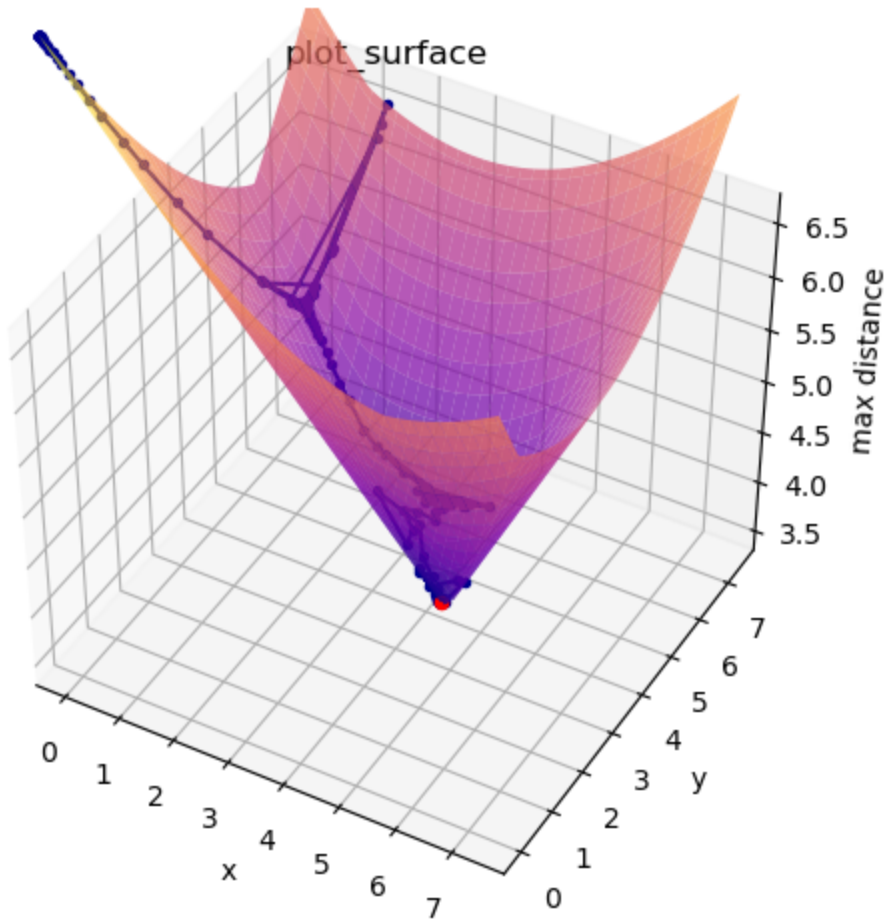
def dist(p, q):
    return ((p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2) ** 0.5
def max_distance(c):
    trace.append(c)
    return max([dist(p, c) for p in zip(x, y)])

c = minimize(max_distance, [0.0, 0.0],
             method='nelder-mead').x
ax = plt.gca()
ax.set_xlim((0, 8))
ax.set_ylim((0, 8))
ax.set_aspect("equal")
plt.plot(x, y, "g.")
plt.plot(*zip(*trace), 'b.-')
ax.add_artist(plt.Circle(c, max_distance(c),
                        color='r', fill=False))
plt.show()
```



Minimum enclosing circle – search space

Maximum distance to an input point



enclosing_circle_search_space.py (previous slide)

```
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

points = [(1.0, 3.0), (3.0, 1.0), (2.5, 3.0),
          (4.0, 6.0), (5.0, 7.0), (6.0, 7.0), (5.0, 2.0)]

# Minimum enclosing circle solver
trace = []

def distance(p, q):
    return ((p[0]-q[0])**2 + (p[1]-q[1])**2)**0.5

def distance_max(q):
    dist = max([distance(p, q) for p in points])
    trace.append((q, dist))
    return dist

solution = minimize(distance_max, [0.0, 0.0],
                    method='nelder-mead')

center = solution.x
radius = solution.fun

# unzip point coordinates
points_x, points_y = zip(*points)
trace_x, trace_y, trace_z = zip(*trace)

# Bounding box [x_min, x_max] x [y_min, y_max]
xs, ys = points_x + trace_x, points_y + trace_y
x_min, x_max = min(xs), max(xs)
y_min, y_max = min(ys), max(ys)
# enforce aspect ratio
x_max = max(x_max, x_min + y_max - y_min)
y_max = max(y_max, y_min + x_max - x_min)
```

```
# Minimum enclosing circle - 3D surface plot
# (plot_surface requires X, Y, Z are 2D numpy.arrays)
X, Y = np.meshgrid(np.linspace(x_min, x_max, 100),
                   np.linspace(y_min, y_max, 100))
Z = np.zeros(X.shape)
for px, py in points:
    Z = np.maximum(Z, (X - px)**2 + (Y - py)**2)
Z = np.sqrt(Z)

ax = plt.subplot(1, 2, 1, projection='3d')
ax.plot_surface(X, Y, Z, cmap='plasma', alpha=0.7)
ax.plot(trace_x, trace_y, trace_z, '.-', c='darkblue')
ax.scatter(*center, radius, 'o', c='red')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('max distance')
ax.set_title('plot_surface')

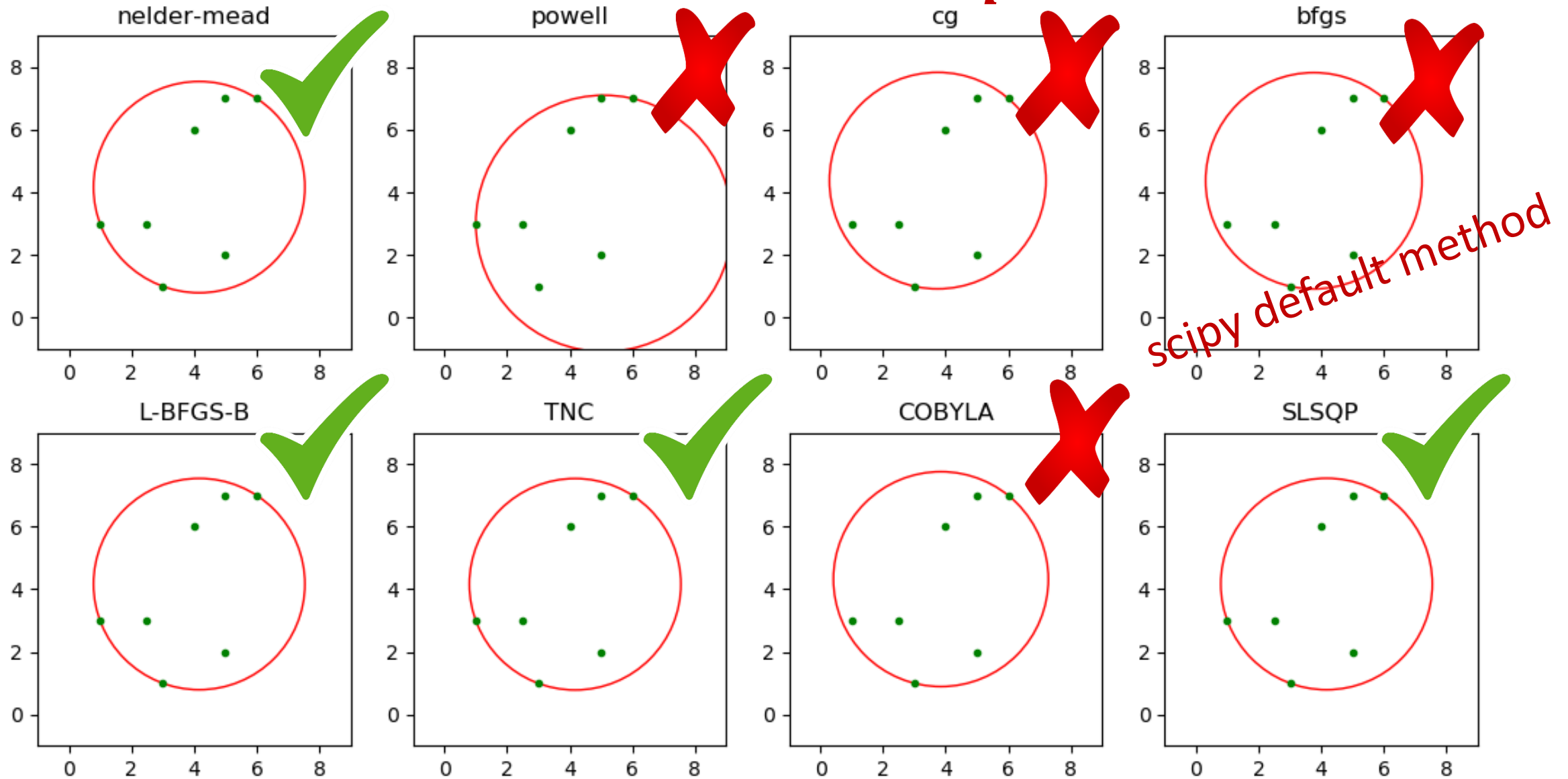
# Minimum enclosing circle - contour plot
plt.subplot(1, 2, 2)
plt.title('pyplot.contour')
plt.plot(trace_x, trace_y, '.-', color='darkblue')
plt.plot(points_x, points_y, 'o', color='darkgreen')
plt.plot(*center, 'o', c='red')
qcs = plt.contour(X, Y, Z, levels=30, cmap='plasma')
plt.clabel(qcs, inline=1, fontsize=8, fmt='%1f')

plt.suptitle('Maximum distance to an input point')
plt.tight_layout()
plt.show()
```

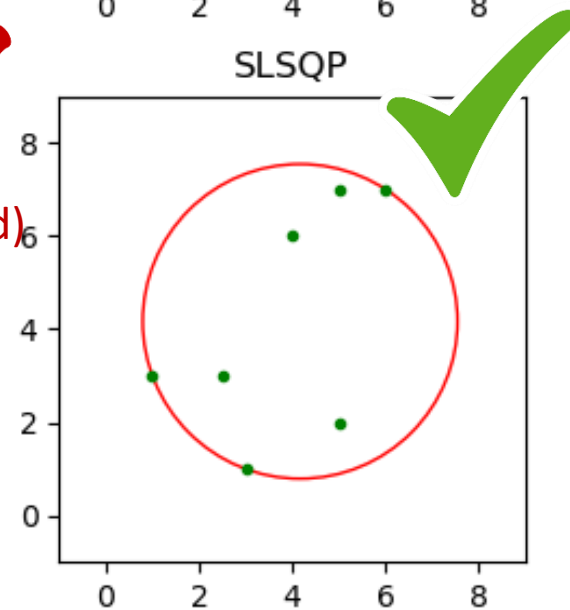
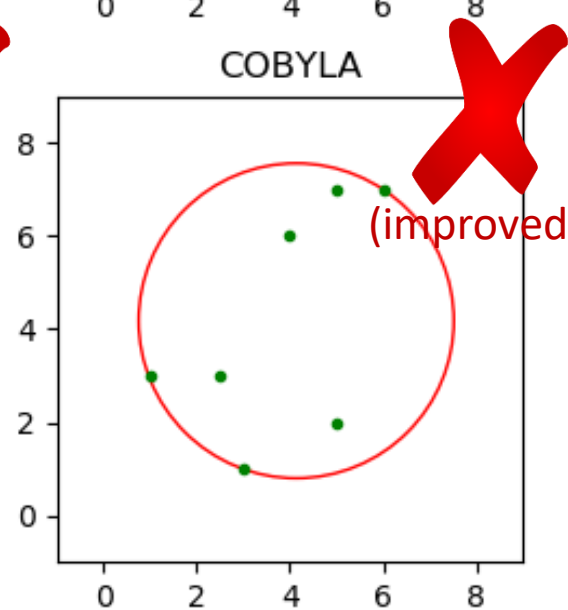
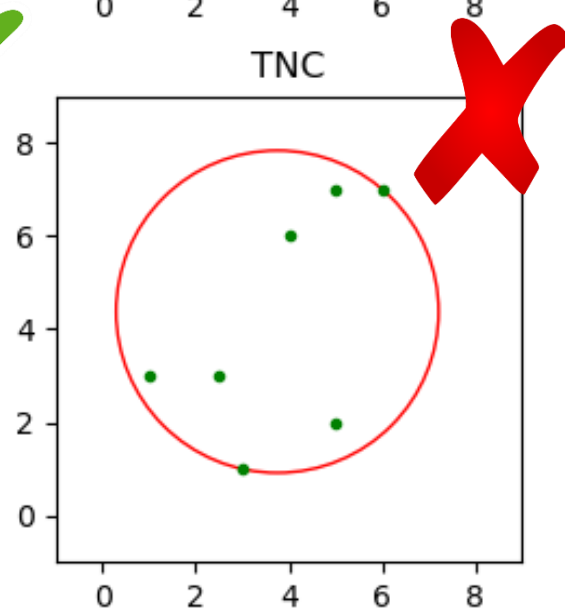
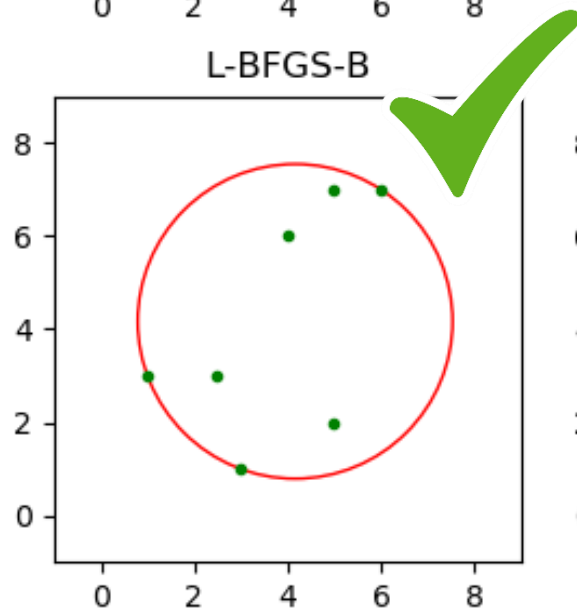
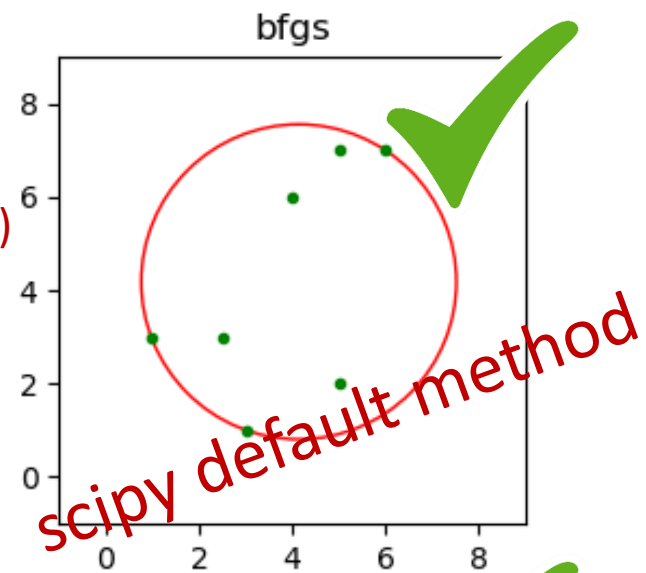
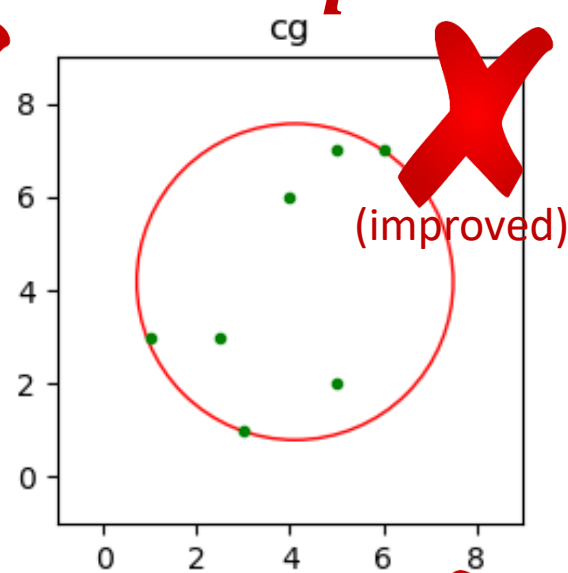
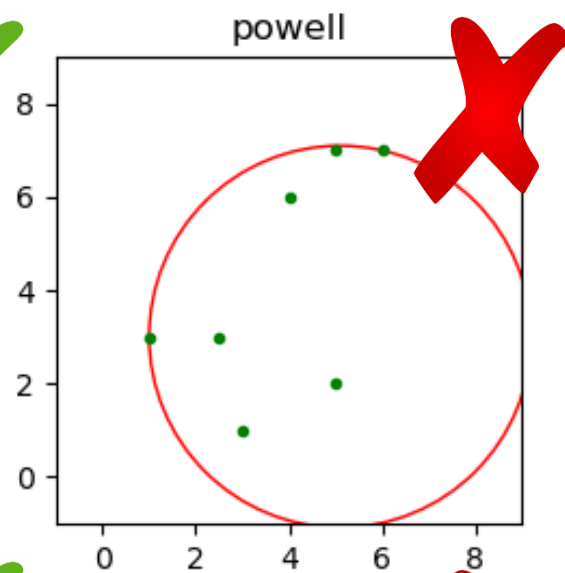
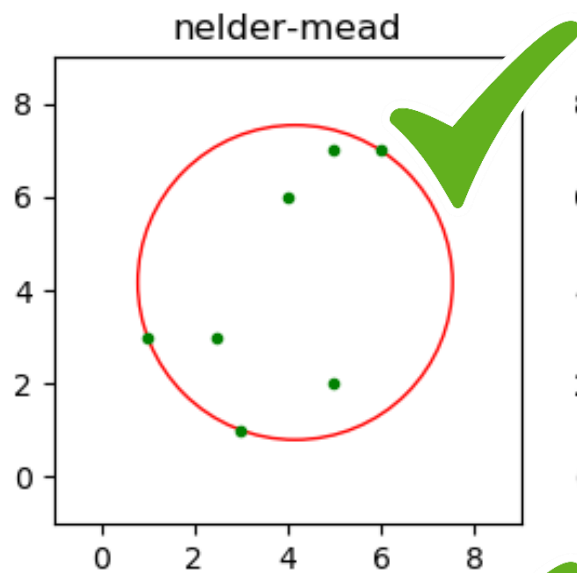
numpy
arrays



scipy.minimize $f(c) = \max_p |p - c|$



scipy.minimize $f(c) = \max_p |p - c|^2$ avoids $\sqrt{\quad}$



Multi-dimensional data

- NumPy
- matrix multiplication, @
- `numpy.linalg.solve`, `numpy.polyfit`

Array programming with NumPy

Harris et al.

Nature, volume 585, pages 357–362, 2020

DOI [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2)



- NumPy is a Python package for dealing with multi-dimensional data

www.numpy.org

docs.scipy.org/doc/numpy/user/quickstart.html

pylab ?

- Guttag [2nd edition] uses pylab in the examples, but...

*“**pylab** is a convenience module that bulk imports **matplotlib.pyplot** (for plotting) and **numpy** (for mathematics and working with arrays) in a single name space. Although many examples use pylab, **it is no longer recommended.**”*

NumPy arrays (example)

Python shell

```
> range(0, 1, .3)
| TypeError: 'float' object cannot be
| interpreted as an integer
> [1 + i / 4 for i in range(5)]
| [1.0, 1.25, 1.5, 1.75, 2.0]
```

} python only supports ranges of int

} generate 5 uniform values in range [1,2]

Python shell

```
> import numpy as np
> np.arange(0, 1, 0.3)
| array([0. , 0.3, 0.6, 0.9])
> type(np.arange(0, 1, 0.3))
| <class 'numpy.ndarray'>
> help(numpy.ndarray)
| +2000 lines of text
> np.linspace(1, 2, 5)
| array([1. , 1.25, 1.5 , 1.75, 2.  ])
```

} numpy can generate ranges with float

} returns a "NumPy array" (not a list)

"arange" $\equiv$ "array range" and generates the array explicitly

} generate n uniformly spaced values

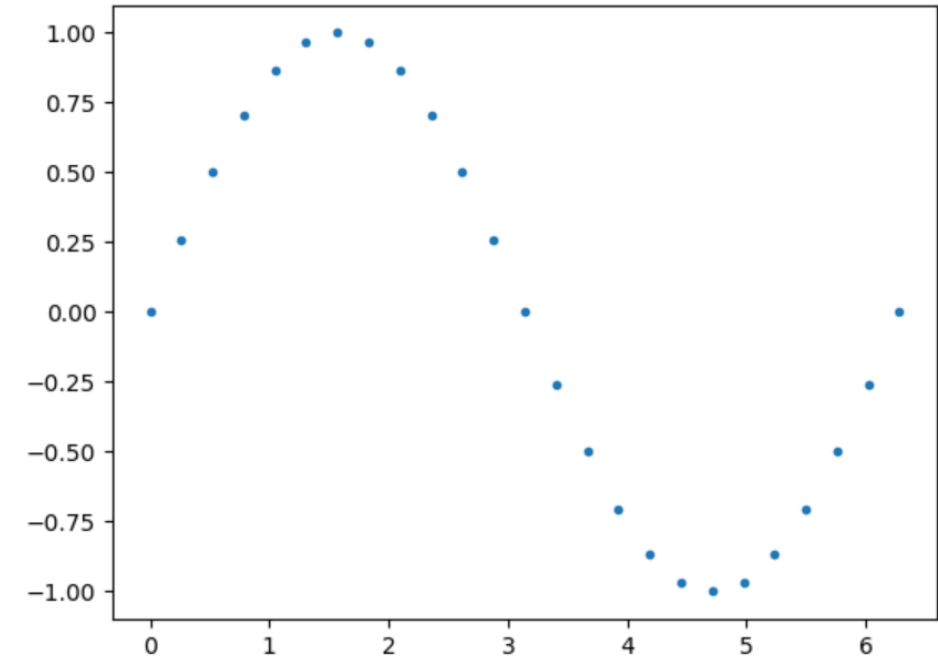
Plotting a function (example)

sin.py

```
import matplotlib.pyplot as plt
import math
n = 25
x = [2 * math.pi * i / (n - 1) for i in range(n)]
y = [math.sin(v) for v in x]
plt.plot(x, y, '.')
plt.show()
```

sin_numpy.py

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2 * np.pi, 25)
y = np.sin(x)
plt.plot(x, y, '.')
plt.show()
```

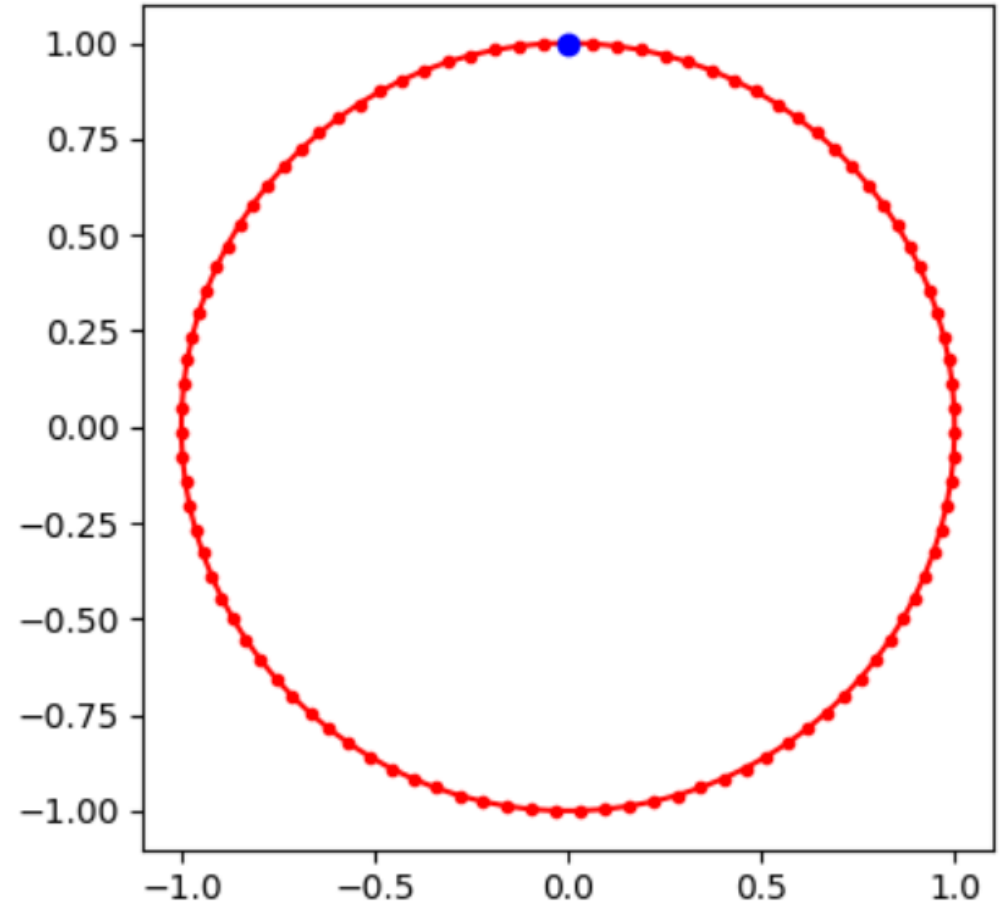


- `np.sin` applies the `sin` function *to each* element of `x`
- `pyplot` accepts NumPy arrays
- `math.pi == np.pi ≈  $\frac{22}{7}$`

A circle

circle.py

```
import matplotlib.pyplot as plt
import numpy as np
a = np.linspace(0, 2 * np.pi, 100)
x = np.sin(a)
y = np.cos(a)
plt.plot(x, y, 'r.-')
plt.plot(x[0], y[0], 'bo')
plt.show()
```



Two half circles

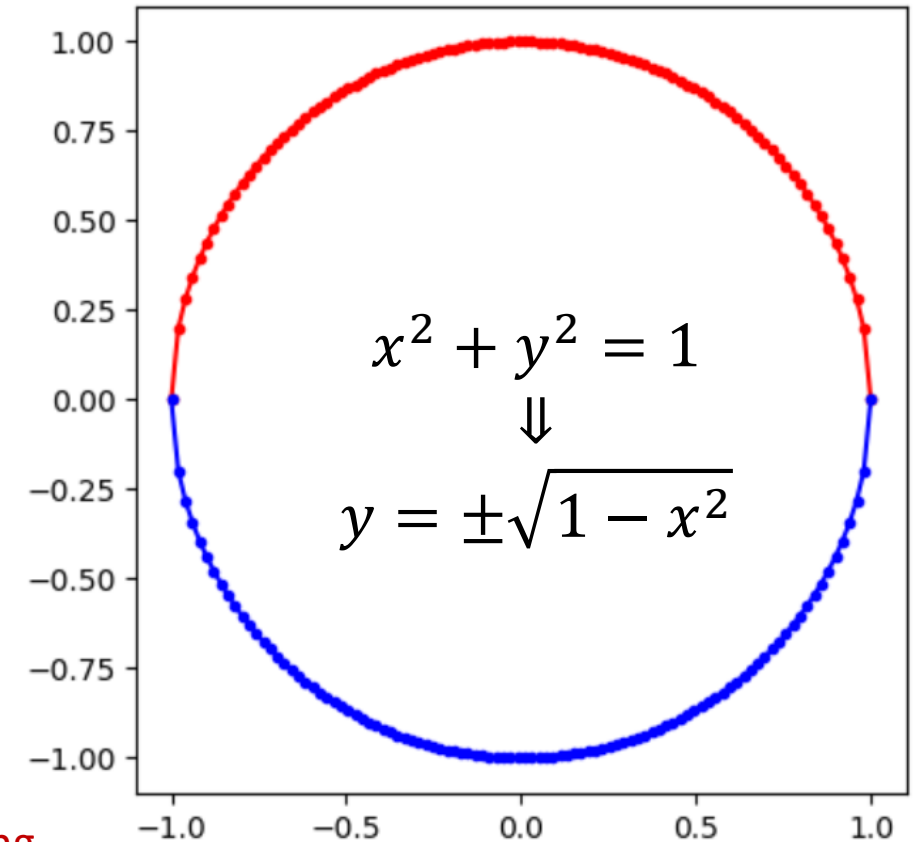
half_circles.py

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-1, 1, 100)
plt.plot(x, np.sqrt(1 - x ** 2), 'r.-')
plt.plot(x, -np.sqrt(1 - x ** 2), 'b.-')
plt.show()
```

compact expression computing
something quite complicated

- `x` is a NumPy array
- `**` NumPy method `__pow__` squaring each element in `x`
- `binary` – NumPy method `__rsub__` that for each element `e` in `x` computes `1 - e`
- `np.sqrt` NumPy method computing the square root of each element in `x`
- `unary` – NumPy method `__neg__` that negates each element in `x`



Creating one-dimensional NumPy arrays

Python shell

```
> np.array([1, 2, 3])
| array([1, 2, 3])
> np.array((1, 2, 3))
| array([1, 2, 3])
> np.array(range(1, 4))
| array([1, 2, 3])
> np.arange(1., 4., 1.)
| array([1., 2., 3.])
> np.linspace(1, 3, 3)
| array([1., 2., 3.])
> np.zeros(3)
| array([0., 0., 0.])
> np.ones(3)
| array([1., 1., 1.])
> np.full(3, 7)
| array([7, 7, 7])
> np.random.random(3)
| array([0.73761651,
| 0.60607355, 0.3614118 ])
```

```
> np.array([1, 2, 3]).dtype # type of all values
| dtype('int32')
> np.arange(3, dtype='float')
| array([0., 1., 2.])
> np.arange(3, dtype='int16') # 16 bit integers
| array([0, 1, 2], dtype=int16)
> np.arange(3, dtype='int32') # 32 bit integers
| array([0, 1, 2])
> 1000 ** np.arange(5)
| array([1, 1000, 1000000, 1000000000, -727379968],
| dtype=int32) # OOPS.. overflow
> 1000 ** np.arange(5, dtype='O')
| array([1, 1000, 1000000, 1000000000,
| 1000000000000], dtype=object) # Python integer
> np.arange(3, dtype='complex')
| array([0.+0.j, 1.+0.j, 2.+0.j])
```



Elements of a NumPy array are not arbitrary precision integers by default – you can select between +25 number representations



slow

Supported dtypes

-  The supported dtypes is platform dependent

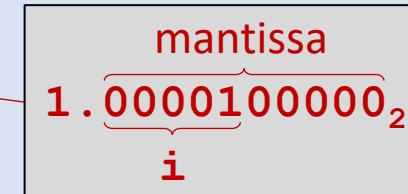
Python shell

```
> import numpy as np
> list(np.sctypeDict)
| ['bool', 'float16', 'float32', 'float64', 'longdouble', 'complex64',
  'complex128', 'clongdouble', 'bytes_', 'str_', 'void', 'object_',
  'datetime64', 'timedelta64', 'int8', 'byte', 'uint8', 'ubyte', 'int16',
  'short', 'uint16', 'ushort', 'intc', 'uintc', 'int32', 'long', 'uint32',
  'ulong', 'int64', 'longlong', 'uint64', 'ulonglong', 'intp', 'uintp',
  'double', 'cdouble', 'single', 'csingle', 'half', 'bool_', 'int_', 'uint',
  'float', 'complex', 'object', 'bytes', 'a', 'int', 'str', 'unicode']
```

Mantissa size in various numpy floats

Python shell

```
> for data_type in ['half', 'float', 'single', 'double', 'longdouble', 'float32', 'float64']:
    x = np.array([1], dtype=data_type)
    for i in range(100):
        if x == x + (x / 2) ** i:
            break
    print(data_type, i - 1, 'bits mantissa')
half 10 bits mantissa
float 52 bits mantissa
single 23 bits mantissa
double 52 bits mantissa
longdouble 52 bits mantissa
float32 23 bits mantissa      # platform independent
float64 52 bits mantissa     # platform independent
```



Creating multi-dimensional NumPy arrays

Python shell

```
> np.array([[1, 2, 3], [4, 5, 6]])
| array([[1, 2, 3],
|        [4, 5, 6]])
> np.array([[1, 2], [4, 5, 6]])
| ValueError: inhomogeneous shape
> np.arange(1, 7).reshape(2, 3)
| array([[1, 2, 3],
|        [4, 5, 6]])
> x = np.arange(12).reshape(2, 2, 3)
> x
| array([[[ 0,  1,  2],
|          [ 3,  4,  5]],
|        [[ 6,  7,  8],
|          [ 9, 10, 11]]])
> numpy.zeros((2, 5), dtype='int32')
| array([[0, 0, 0, 0, 0],
|        [0, 0, 0, 0, 0]])
```



```
> x.size
| 12
> x.ndim
| 3
> x.shape
| (2, 2, 3)
> x.dtype
| dtype('int32')
> x.flatten()
| array([0,1,2,3,4,5,6,7,8,9,10,11])
> list(x.flat) # flat is an iterator
| [0,1,2,3,4,5,6,7,8,9,10,11]
> np.eye(3) # diagonal 2D array
| array([[1., 0., 0.],
|        [0., 1., 0.],
|        [0., 0., 1.]])
```

View vs Copy

- Numpy is optimized to handle big multi-dimensional arrays
- To *avoid copying* data, **views** allows one to look at the underlying data in different ways (data can be shared by multiple views)
- `reshape`, `ravel` and slicing are examples creating views
- `flatten` and `ravel` both turn multiple dimensional arrays into one dimensional arrays but `flatten` creates an explicit copy whereas `ravel` creates a space efficient view

Python shell

```
> x = np.arange(6)
> y = x.reshape(2, 3) # view
> y[0][0] = 42        # updates x
> x
| array([42, 1, 2, 3, 4, 5])
> y
| array([[42, 1, 2],
|        [ 3, 4, 5]])
> z = y.flatten()      # copy
> z[5] = 0
> z
| array([42, 1, 2, 3, 4, 0])
> x
| array([42, 1, 2, 3, 4, 5])
> w = y.ravel()         # view
> w[5] = -1
> w
| array([42, 1, 2, 3, 4, -1])
> x
| array([42, 1, 2, 3, 4, -1])
```

NumPy operations

Python shell

```
> x = numpy.arange(3)
> x
| array([0, 1, 2])
> x + x # elementwise addition
| array([0, 2, 4])
> 1 + x # add integer to each element
| array([1, 2, 3])
> x * x # elementwise multiplication
| array([0, 1, 4])
> np.dot(x, x) # dot product
| 5
> np.cross([1, 2, 3], [3, 2, 1]) # cross product
| array([-4, 8, -4])
```

```
> a = np.arange(6).reshape(2,3)
> a
| array([[0, 1, 2],
|        [3, 4, 5]])
> a.T # matrix transposition, view
| array([[0, 3],
|        [1, 4],
|        [2, 5]])
> a @ a.T # matrix multiplication
| array([[ 5, 14],
|        [14, 50]])
> a += 1 # modifies view
> a
| array([[1, 2, 3],
|        [4, 5, 6]])
```

Universal functions (apply to each entry)

Python shell

```
> x = np.array([[1, 2], [3, 4]])
> np.sin(x) # also: cos, exp, sqrt, log, ceil, floor, abs
| array([[ 0.84147098,  0.90929743],
|         [ 0.14112001, -0.7568025 ]])
> np.sign(np.sin(x))
| array([[ 1.,  1.],
|         [ 1., -1.]])
> np.mod(np.arange(10), 3) # same as: np.arange(10) % 3
| array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype=int32)
```

Axis

Python shell

```
> x = np.arange(1, 7).reshape(2, 3)
> x
| array([[1, 2, 3],
|        [4, 5, 6]])
> x.sum()    # = x.sum(axis=(0, 1))
| 21
> x.sum(axis=0)
| array([5, 7, 9])
> x.sum(axis=1)
| array([6, 15])
> x.min()    # = x.min(axis=(0, 1))
| 1
```

Python shell

```
> x.min(axis=0)
| array([1, 2, 3])
> x.min(axis=1)
| array([1, 4])
> x.cumsum()
| array([ 1,  3,  6, 10, 15, 21], dtype=int32)
> x.cumsum(axis=0)
| array([[1, 2, 3],
|        [5, 7, 9]], dtype=int32)
> x.cumsum(axis=1)
| array([[ 1,  3,  6],
|        [ 4,  9, 15]], dtype=int32)
```

Slicing

Python shell

```
> x = numpy.arange(20).reshape(4, 5)
> x
| array([[ 0,  1,  2,  3,  4],
|        [ 5,  6,  7,  8,  9],
|        [10, 11, 12, 13, 14],
|        [15, 16, 17, 18, 19]])
> x[2, 3] # = x[(2, 3)]
| 13
> x[1:4:2, 2:4:1] # rows 1 and 3, and columns 2 and 3, view
| array([[ 7,  8],
|        [17, 18]])
> x[:, 3]
| array([ 3,  8, 13, 18])
> x[..., 3] # ... is placeholder for ':' for all missing dimensions
| array([ 3,  8, 13, 18])
> type(...)
| <class 'ellipsis'>
```


Broadcasting (stretching arrays to get same size)

- Numpy tries to apply *broadcasting*, if array shapes do not match, i.e. adds missing leading dimensions and repeats a dimension with only one element:

```
[[1],[2]] + [10,20]    column + row vector
≡ [[1],[2]] + [[10,20]] both ndim = 2
≡ [[1,1],[2,2]] + [[10,20]]
≡ [[1,1],[2,2]] + [[10,20],[10,20]]
≡ [[11,21],[12,22]]
```

- To prevent unexpected broadcasting, add an assertion to your program:



```
assert x.shape == y.shape
```

Python shell

```
> x = np.array([[1, 2, 3], [4, 5, 6]])
> y = np.array([1, 2, 3]) # one row
> z = np.array([[1], [2]]) # one column
> x + 3 # add 3 to each entry
| array([[4, 5, 6],
|        [7, 8, 9]])
> x + y # add y to each row
| array([[2, 4, 6],
|        [5, 7, 9]])
> x + z # add z to each column
| array([[2, 3, 4],
|        [6, 7, 8]])
> y + z # 2 rows with y + 3 columns with z
| array([[2, 3, 4],
|        [3, 4, 5]])
> z == z.T # [[1,1],[2,2]] == [[1,2],[1,2]]
| array([[ True, False],
|        [False,  True]])
```

Masking

Python shell

```
> x = np.arange(1, 11).reshape(2, 5)
> x
| array([[ 1,  2,  3,  4,  5],
|        [ 6,  7,  8,  9, 10]])
> x % 3
| array([[1, 2, 0, 1, 2],
|        [0, 1, 2, 0, 1]], dtype=int32)
> x % 3 == 0
| array([[False, False,  True, False, False],
|        [ True, False, False,  True, False]])
> x[x % 3 == 0] # use Boolean matrix to select entries
| array([3, 6, 9])
> x[:, x.sum(axis=0) % 3 == 0] # columns with sum divisible by 3
| array([[ 2,  5],
|        [ 7, 10]])
```

Numpy is fast... but be aware of dtype



Python shell



```
> sum([x**2 for x in range(1000000)])
| 333332833333500000
> (np.arange(1000000)**2).sum()
| 584144992 # wrong since overflow when default dtype='int32'
> (np.arange(1000000, dtype='int64')**2).sum()
| 333332833333500000 # 64 bit integers do not overflow
> import timeit from timeit
> timeit('sum([x**2 for x in range(1000000)])', number=1)
| 0.5614346340007614

> timeit('(np.arange(1000000)**2).sum()', setup='import numpy as np', number=1)
| 0.014362967000124627 # ridiculous fast but also wrong result...
> timeit('(np.arange(1000000, dtype='int64')**2).sum()',
| setup='import numpy as np', number=1)
| 0.048017077999247704 # fast and correct
> np.iinfo(np.int32).min
| -2147483648
> np.iinfo(np.int32).max
| 2147483647
```

numpy.int32 – 32 bit signed two's-complement integers

32 bits $b_{31}b_{30}b_{29}b_{28}\cdots b_2b_1b_0$
represent the value

$$-b_{31} \cdot 2^{31} + \sum_{i=0}^{30} b_i \cdot 2^i$$

10000000000000000000000000000000	-2147483648
100000000000000000000000000000001	-2147483647
⋮	
11111111111111111111111111111110	-2
11111111111111111111111111111111	-1
00000000000000000000000000000000	0
000000000000000000000000000000001	1
000000000000000000000000000000010	2
⋮	
01111111111111111111111111111110	2147483646
01111111111111111111111111111111	2147483647

Note: There is one more negative number than positive

Python shell

```
> np.int32(- 2 ** 31)
| -2147483648
> np.int32(- 2 ** 31) + 1
| -2147483647
> np.int32(- 2 ** 31) - 1
| 2147483647
> np.int32(2 ** 31)
| OverflowError: Python int too
  large to convert to C long
> np.int32(2 ** 31 - 1)
| 2147483647
> np.int32(2 ** 31 - 1) + 1
| -2147483648
> np.abs(np.int32(-2147483647))
| 2147483647
> np.abs(np.int32(-2147483648))
| -2147483648
```



numpy.org/doc/stable/user/basics.types.html

en.wikipedia.org/wiki/Two's_complement

Linear algebra

Python shell

```
> x = np.arange(1, 5, dtype=float).reshape(2, 2)
> x
| array([[1., 2.],
|        [3., 4.]])
> x.T # matrix transpose
| array([[1., 3.],
|        [2., 4.]])
> np.linalg.det(x) # matrix determinant
| -2.0000000000000004
> np.linalg.inv(x) # matrix inverse
| array([[-2. ,  1. ],
|        [ 1.5, -0.5]])
> np.linalg.eig(x) # eigenvalues and eigenvectors
| (array([-0.37228132,  5.37228132]),
|  array([[-0.82456484, -0.41597356], [0.56576746, -0.90937671]]))
> y = np.array([[5.], [7.]])
> np.linalg.solve(x, y) # solve linear matrix equations
| array([[-3.],
|        [ 4.]]) # z1
|                # z2
```

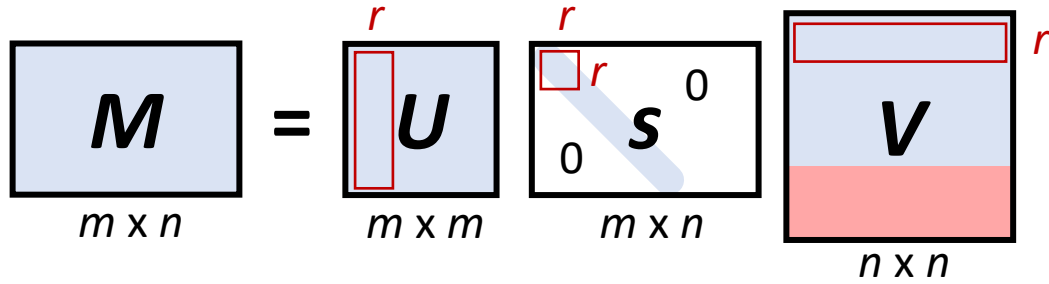


numpy.matrix

"It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future."

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \end{pmatrix}$$

Singular value decomposition, `np.linalg.svd`



- U and V unitary matrix ($UU^T = I$)
- S diagonal matrix, decreasing singular values

`np.ndarray shape=(520, 800) dtype=uint8 min=0 max=252`

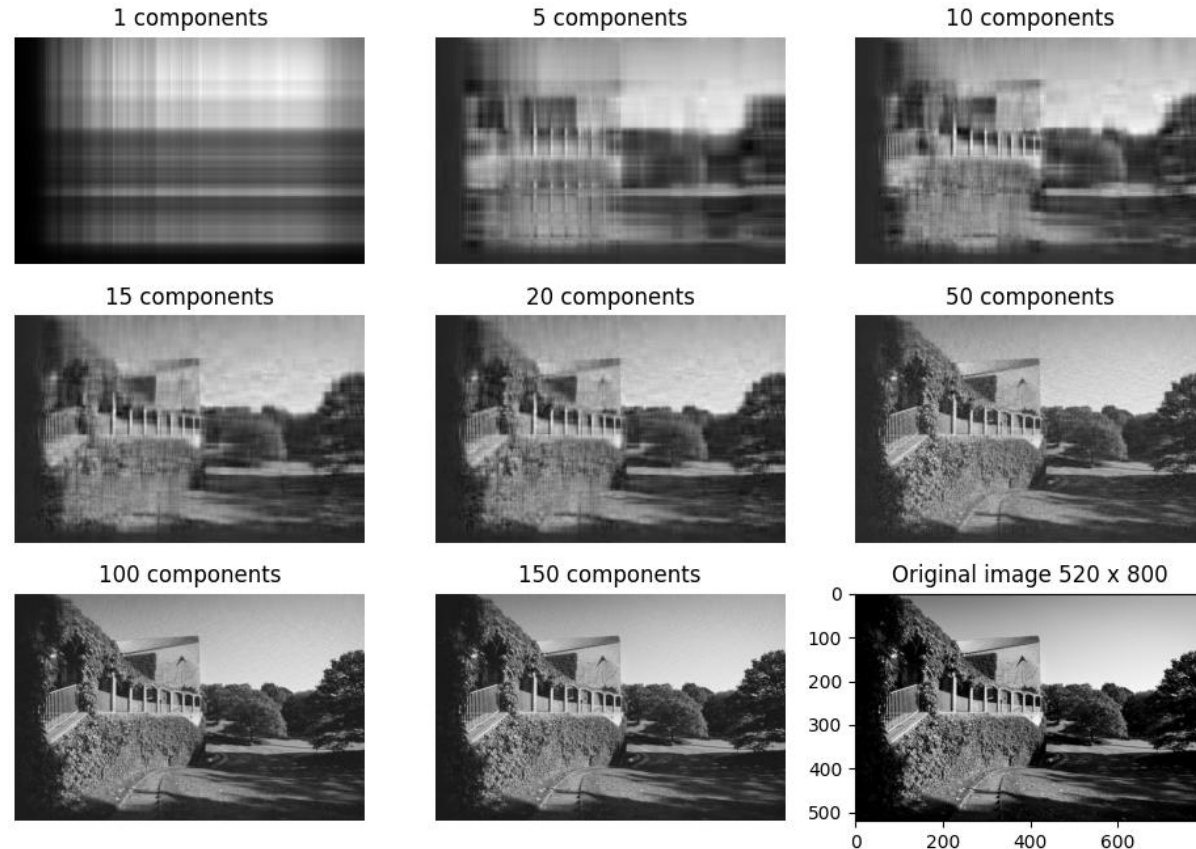
image-reconstruction.py

```
import numpy as np
import cv2 # Computer Vision, opencv.org
import matplotlib.pyplot as plt

color = cv2.imread('university.jpg') # color image
gray = cv2.cvtColor(color, cv2.COLOR_BGR2GRAY) # convert to gray

u, s, v = np.linalg.svd(gray) # Calculating the SVD

for i, r in enumerate([1, 5, 10, 15, 20, 50, 100, 150], start=1):
    rank_r = u[:, :r] @ np.diag(s[:r]) @ v[:, r, :]
    plt.subplot(3, 3, i)
    plt.imshow(rank_r, cmap='gray', vmin=0, vmax=255)
    plt.title(f'{r} components')
    plt.axis('off')
plt.subplot(3, 3, 9)
plt.imshow(gray, cmap='gray')
plt.title(f'Original image {gray.shape[0]} x {gray.shape[1]}')
plt.show()
```



... and in color

image-reconstruction-color.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread

color = imread('university.jpg')
color = color / 255 # convert integers 0..255 to floats 0..1

plt.subplot(4, 2, 8)
plt.imshow(color)
plt.axis('off')
plt.title(f'Original')

height, width, colors = color.shape

u, s, v = np.linalg.svd(color.reshape((height, width * colors)),
                        full_matrices=False)

for i, r in enumerate([1, 2, 5, 10, 25, 50, 125], start=1):
    rank_r = u[:, :r] @ np.diag(s[:r]) @ v[:, r, :]
    plt.subplot(4, 2, i)
    plt.imshow(rank_r.reshape((height, width, colors)))
    plt.title(f'{r} components')
    plt.axis('off')
plt.show()
```

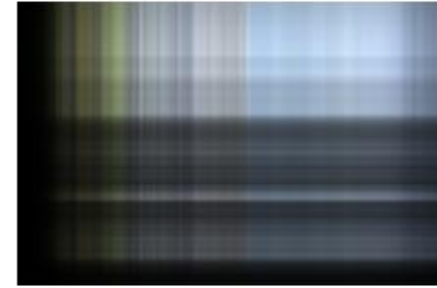
shape=(520, 800, 3)
3 = (red, blue, green)

shape=(520, 2400)

v shape=(520, 2400)
instead of (2400, 2400)

shape=(520, 800, 3)

1 components



2 components



5 components



10 components



25 components



50 components



125 components



Original



... and in color (stacked)

image-reconstruction-color-stacked.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread

color = imread('university.jpg')
color = color / 255 # convert integers 0..255 to floats 0..1

plt.subplot(4, 2, 8)
plt.imshow(color)
plt.axis('off')
plt.title(f'Original')

u, s, v = np.linalg.svd(color.transpose(2, 0, 1), full_matrices=False)
print(f'{u.shape=} {s.shape=} {v.shape=}')

for i, r in enumerate([1, 2, 5, 10, 25, 50, 125], start=1):
    rank_r = (u[:, :, :r] * s[:, None, :r]) @ v[:, :r, :]
    plt.subplot(4, 2, i)
    plt.imshow(rank_r.transpose(1, 2, 0))
    plt.title(f'{r} components')
    plt.axis('off')
plt.show()
```

Annotations:

- shape = (520, 800, 3) points to `color`
- shape = (3, 520, 800) points to `color.transpose(2, 0, 1)`
- element-wise multiplication (*), broadcasting (None), stacked for each color (:) points to `s[:, None, :r]`
- shape = (520, 800, 3) points to `rank_r.transpose(1, 2, 0)`

Python shell

```
| u.shape=(3, 520, 520) s.shape=(3, 520) v.shape=(3, 520, 800)
```

1 components



2 components



5 components



10 components



25 components



50 components



125 components



Original



matplotlib.image.imread



cv2.imread



cv2.imread corrected



color-image.py

```
import matplotlib.pyplot as plt
import matplotlib.image
import cv2

img1 = matplotlib.image.imread('university.jpg')
img2 = cv2.imread('university.jpg') # cv2 uses BGR instead of RGB
img3 = img2[:, :, ::-1]             # change color order BGR to RGB
Images = [(img1, 'matplotlib.image.imread'), (img2, 'cv2.imread'), (img3, 'cv2.imread corrected')]

for i, (img, title) in enumerate(images, start=1):
    plt.subplot(1, 3, i)
    plt.imshow(img)
    plt.axis('off')
    plt.title(title)

plt.show()
```

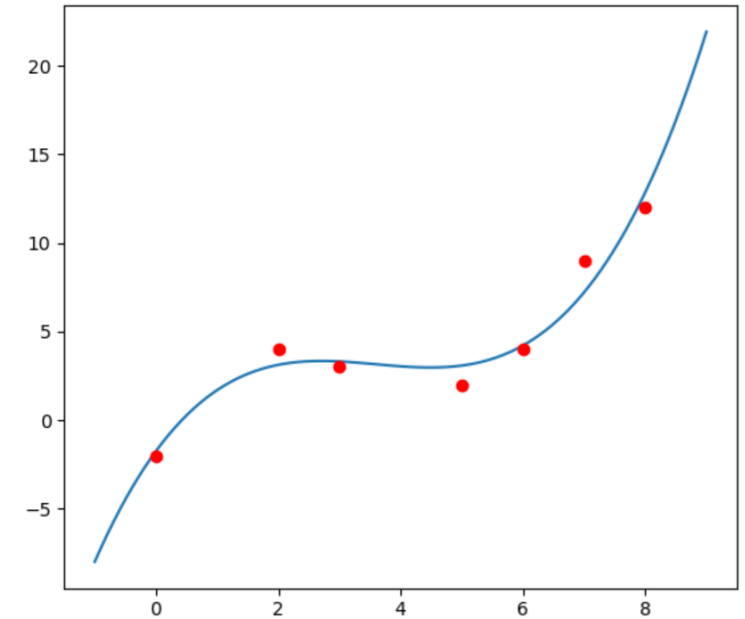


numpy.polyfit

- Given n points with $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$
- Find polynomial p of degree d that minimizes

$$\sum_{i=0}^{n-1} (y_i - p(x_i))^2$$

- known as least squares fit / linear regression / polynomial regression



fit.py

```
import matplotlib.pyplot as plt
import numpy as np

x = [0, 2, 3, 5, 6, 7, 8]
y = [-2, 4, 3, 2, 4, 9, 12]

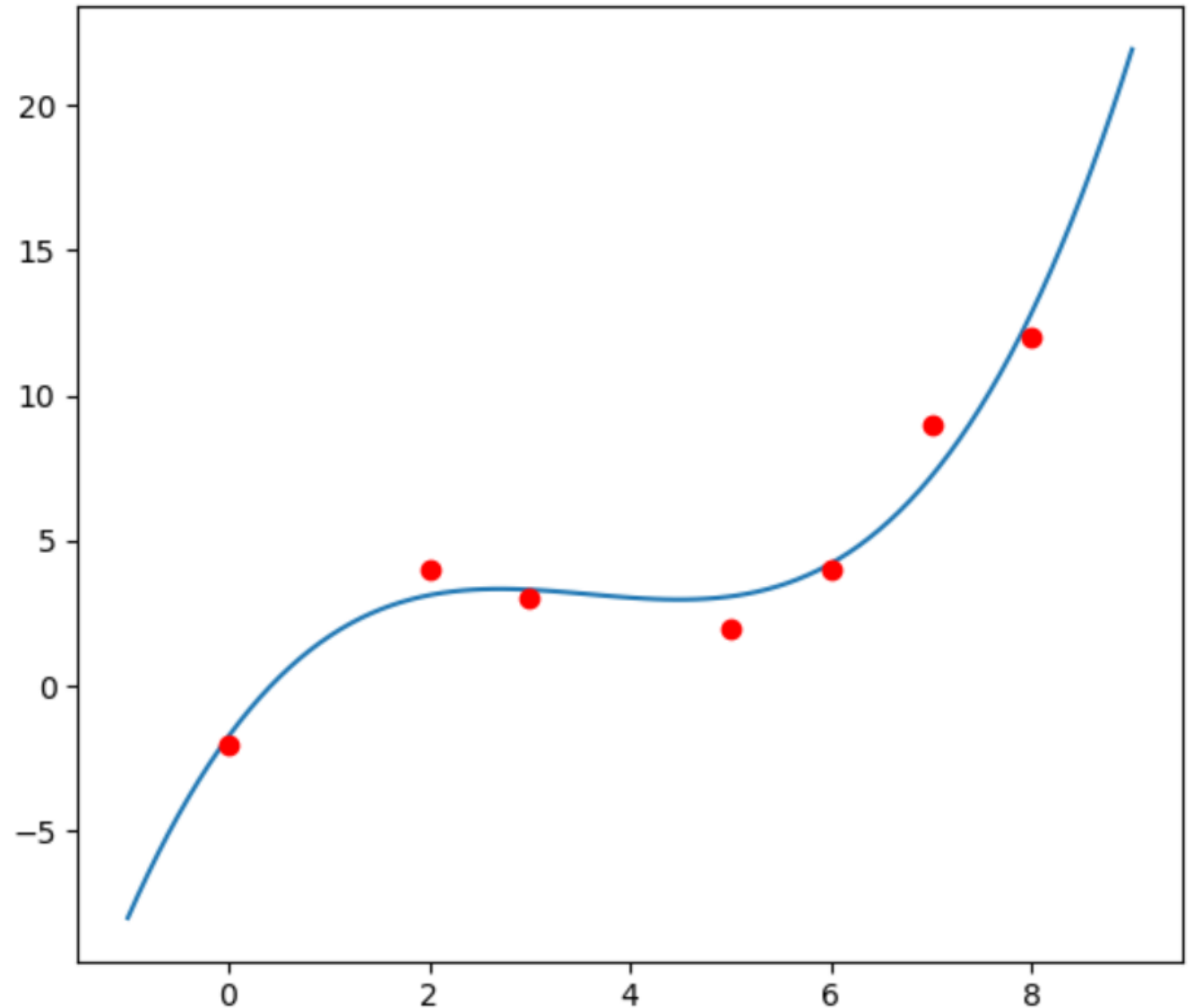
coefficients = np.polyfit(x, y, 3)

fx = np.linspace(-1, 9, 100)
fy = np.polyval(coefficients, fx)

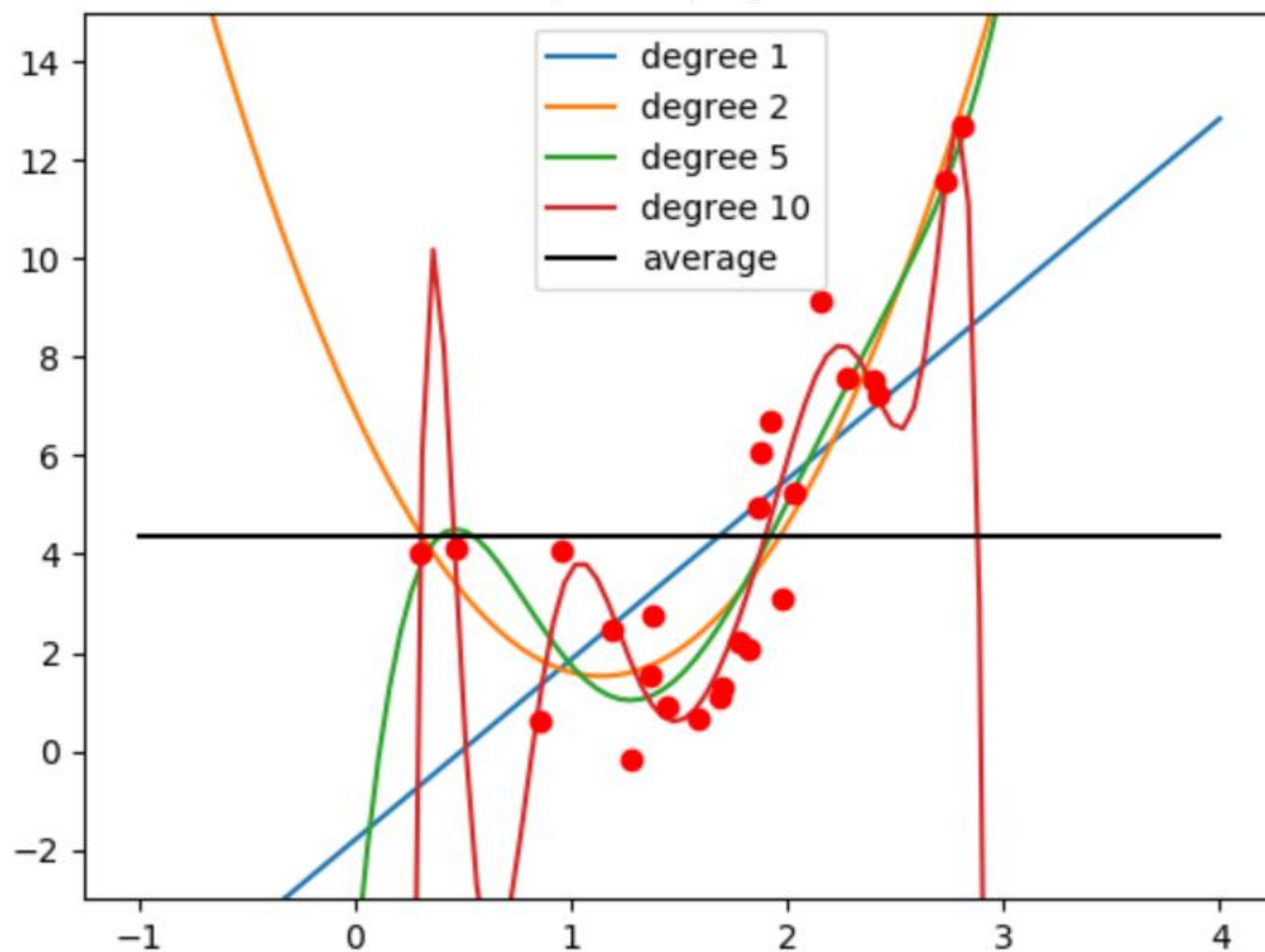
plt.plot(fx, fy, '-')
plt.plot(x, y, 'ro')

plt.show()
```

degree
↓



Least squares polynomial fit



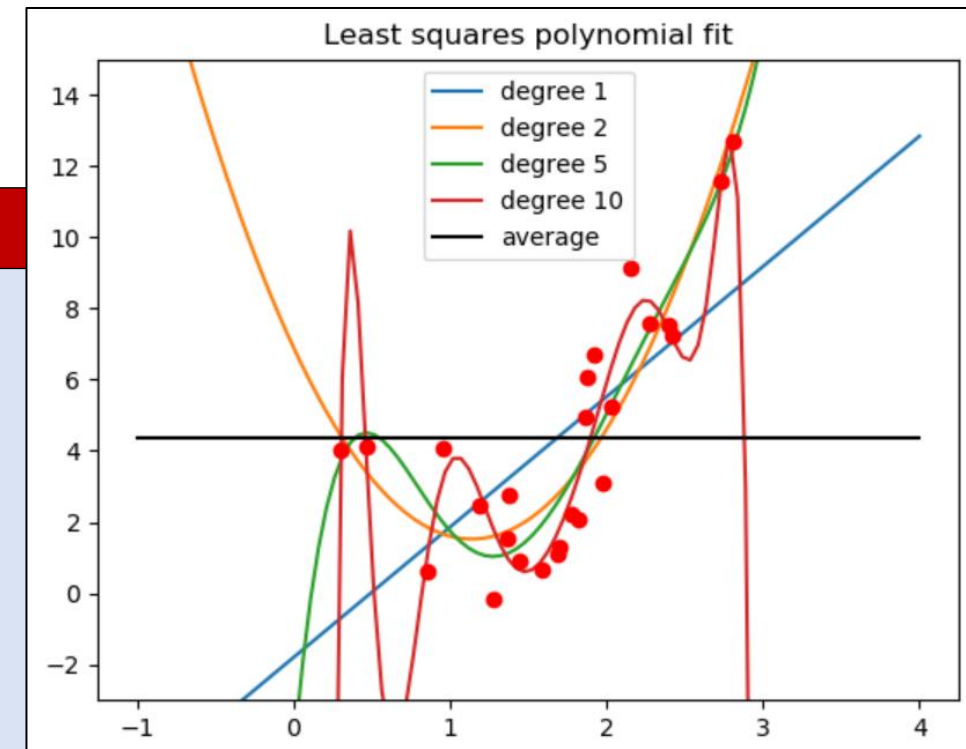
polyfit.py

```
import matplotlib.pyplot as plt
import numpy as np

x = 3 * np.random.random(25)
noise = np.random.random(x.size) ** 2
y = 5 * x ** 2 - 12 * x + 7 + 5 * noise

for degree in [1, 2, 5, 10]:
    coefficients = np.polyfit(x, y, degree)
    fx = np.linspace(-1, 4, 100)
    fy = np.polyval(coefficients, fx)
    plt.plot(fx, fy, '-', label=f'degree {degree}')

avg = np.average(y)
plt.plot(x, y, 'ro')
plt.plot([-1, 4], [avg, avg], 'k-', label='average')
plt.ylim(-3, 15)
plt.title('Least squares polynomial fit')
plt.legend()
plt.show()
```

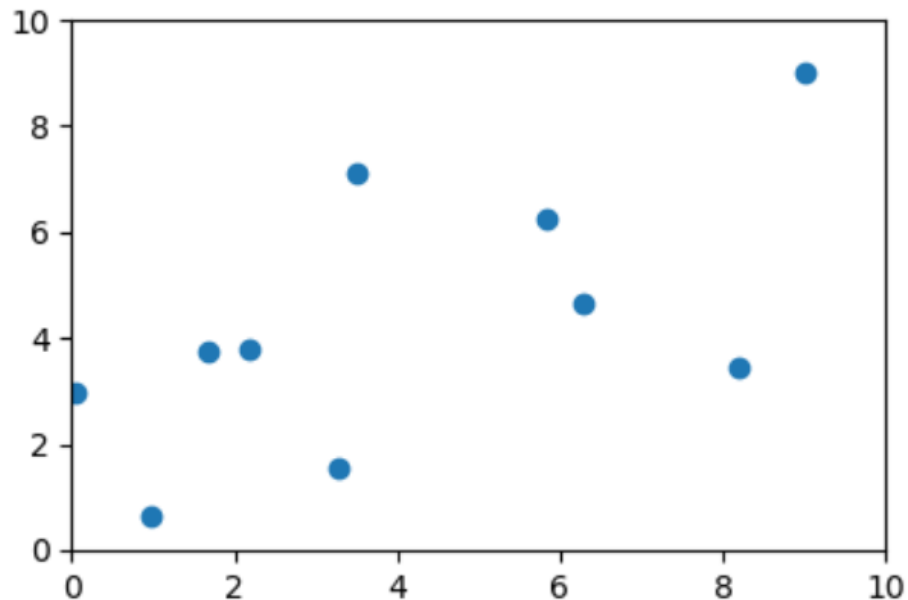


Animating bouncing balls

- matplotlib figures can be animated using

`matplotlib.animation.FuncAnimation`

that as arguments takes the figure to be updated/redrawn, a function to call for each update, and an interval in milliseconds between updates



`balls.py`

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from numpy import zeros, maximum, minimum
from numpy.random import random

g = 0.01
N = 10
x, y = 10.0 * random(N), 1.0 + 9.0 * random(N)
dx, dy = random(N) / 5, zeros(N)

fig = plt.figure()
plt.xlim(0, 10)
plt.ylim(0, 10)
balls, = plt.plot(x, y, 'o') # returns Line2D obj

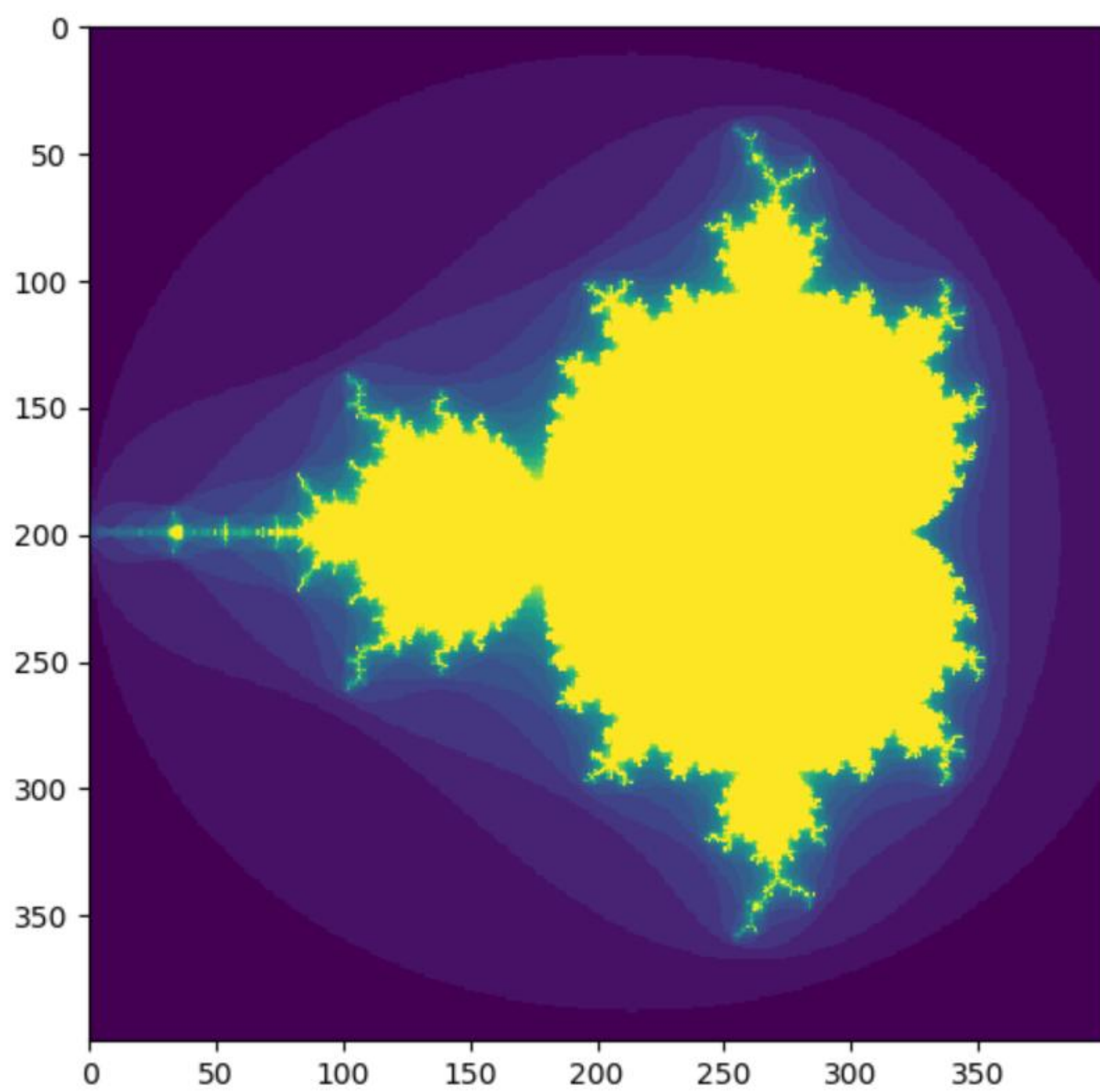
def move(frame):
    global x, y, dx, dy

    x += dx
    bounce = (x > 10.0) | (x < 0.0) # numpy mask
    dx[bounce] = -dx[bounce]
    x = minimum(10.0, maximum(0.0, x))

    y += dy
    bounce = y < 0.0 # numpy mask
    y[bounce] -= dy[bounce]
    dy[bounce] = -dy[bounce]
    dy -= g

    balls.set_data(x, y) # update positions

# removing 'ani =' causes program to fail...
ani = FuncAnimation(fig, move, interval=25)
plt.show()
```

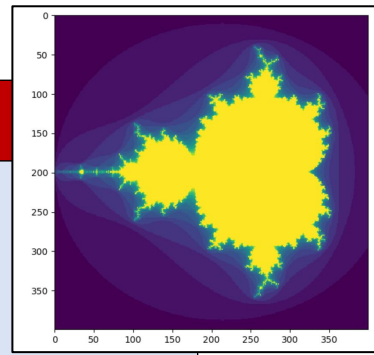



mandelbrot.py

```
import numpy as np
import matplotlib.pyplot as plt

def mandelbrot(h, w, maxit=20):
    '''Returns an image of the Mandelbrot fractal of size (h, w).'''
    x = np.linspace(-2.0, 0.8, w).reshape(1, w) # row vector
    y = np.linspace(-1.4, 1.4, h).reshape(h, 1) # column vector
    c = x + y * 1j # broadcast & complex
    z = c
    divtime = np.full(z.shape, maxit, dtype=int) # all values = maxit
    for i in range(maxit):
        z = z * z + c # elementwise
        diverge = z * np.conj(z) > 4 # who is diverging
        div_now = diverge & (divtime == maxit) # who is diverging now
        divtime[div_now] = i # note when
        z[diverge] = 0 # limit divergence
    return divtime # (avoids overflows)

plt.imshow(mandelbrot(400, 400))
plt.show()
```



Linear programming

- Example Numpy: PageRank
- `scipy.optimize.linprog`
- Example linear programming: Maximum flow

PageRank

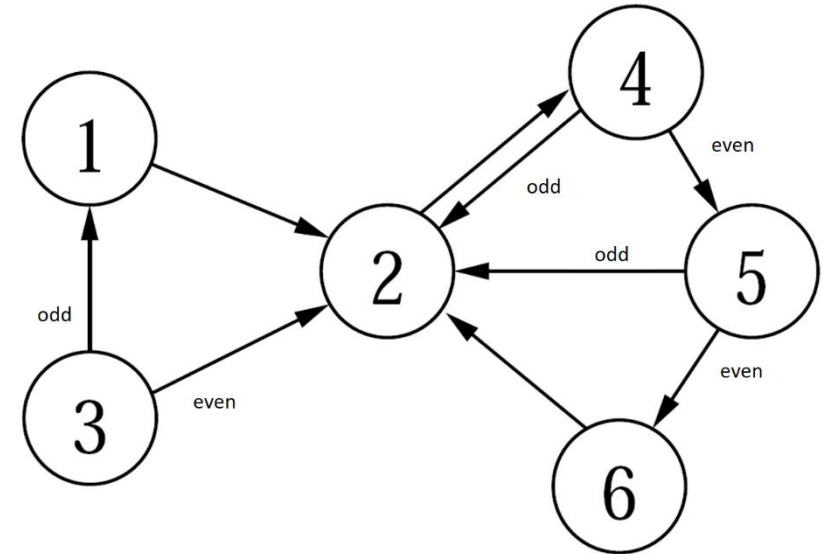
PageRank - A NumPy / Jupyter / matplotlib example

- Google's original search engine ranked webpages using **PageRank**
- View the internet as a graph where **nodes** correspond to webpages and **directed edges** to links from one webpage to another webpage
- Google's PageRank algorithm was described in (infolab.stanford.edu/pub/papers/google.pdf, 1998)

The Anatomy of a Large-Scale Hypertextual Web Search Engine

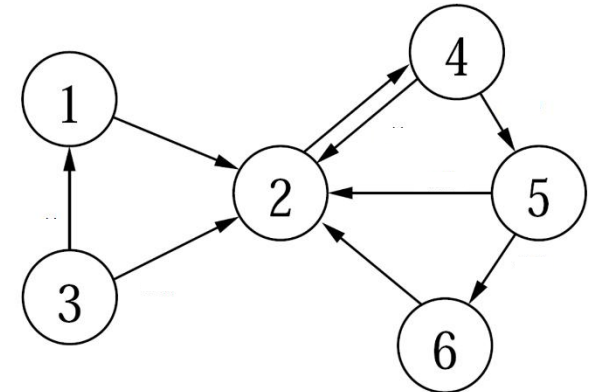
Sergey Brin and Lawrence Page

*Computer Science Department,
Stanford University, Stanford, CA 94305, USA
sergey@cs.stanford.edu and page@cs.stanford.edu*



Five different ways to compute PageRank probabilities

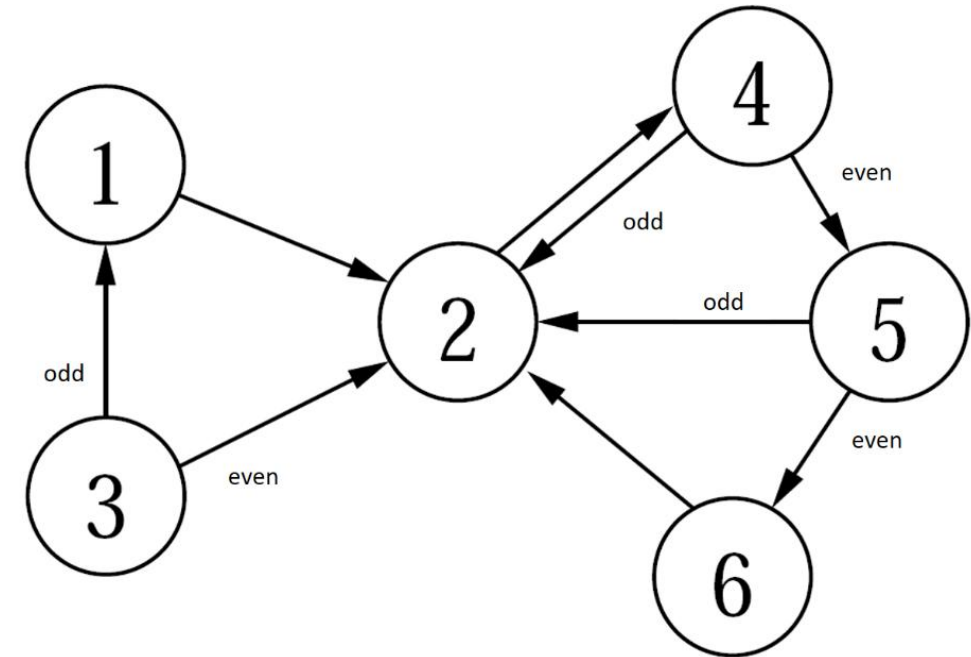
- 1) Simulate random process manually by rolling dices
- 2) Simulate random process in Python
- 3) Computing probabilities using matrix multiplication
- 4) Repeated matrix squaring
- 5) Eigenvector for $\lambda = 1$



Random surfer model (simplified)

The PageRank of a node (web page) is the fraction of the time one visits a node by performing an *infinite random traversal* of the graph starting at node 1, and in each step

- with **probability 1/6** jumps to a **random page** (probability 1/6 for each node)
- with **probability 5/6** follows an **outgoing edge** to an adjacent node (selected uniformly)



The above can be simulated by using a dice: Roll a *dice*. If it shows 6, jump to a random page by rolling the dice again to figure out which node to jump to. If the dice shows 1-5, follow an outgoing edge - if two outgoing edges roll the dice again and go to the lower number neighbor if it is odd.

Adjacency matrix and degree vector

pagerank.ipynb

```
import numpy as np

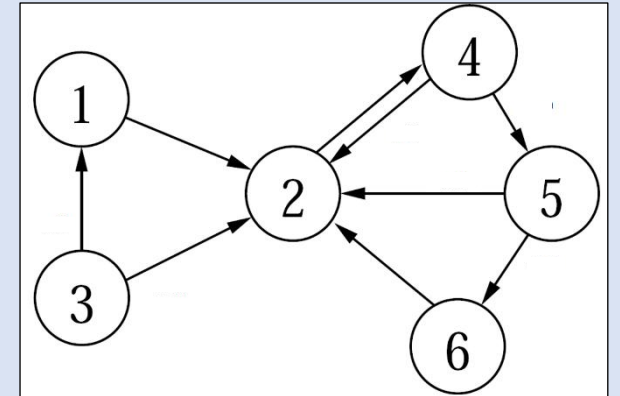
# Adjacency matrix of the directed graph in the figure
# (note that the rows/columns are 0-indexed, whereas in the figure the nodes are 1-indexed)

G = np.array([[0, 1, 0, 0, 0, 0],
              [0, 0, 0, 1, 0, 0],
              [1, 1, 0, 0, 0, 0],
              [0, 1, 0, 0, 1, 0],
              [0, 1, 0, 0, 0, 1],
              [0, 1, 0, 0, 0, 0]])

n = G.shape[0] # number of rows in G
degree = np.sum(G, axis=1, keepdims=True) # column vector with row sums = out-degrees

# The below code handles sinks, i.e. nodes with outdegree zero (no effect on the graph above)

G = G + (degree == 0) # add edges from sinks to all nodes (uses broadcasting)
degree = np.sum(G, axis=1, keepdims=True)
```



Simulate random walk (random surfer model)

pagerank.ipynb

```
from random import randint, choice

STEPS = 1000000

# adjacency_list[i] is a list of all j where (i, j) is an edge of the graph.
adjacency_list = [[j for j, e in enumerate(row) if e] for row in G]

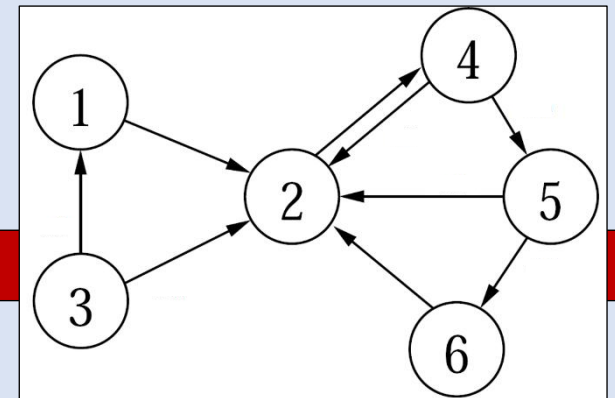
count = np.zeros(n)          # histogram over number of node visits
state = 0                    # start at node with index 0

for _ in range(STEPS):
    count[state] += 1        # increment count for state
    if randint(1, 6) == 6:   # original paper uses 15% instead of 1/6
        state = randint(0, 5)
    else:
        state = choice(adjacency_list[state])

print(adjacency_list, count / STEPS, sep='\n')
```

Python shell

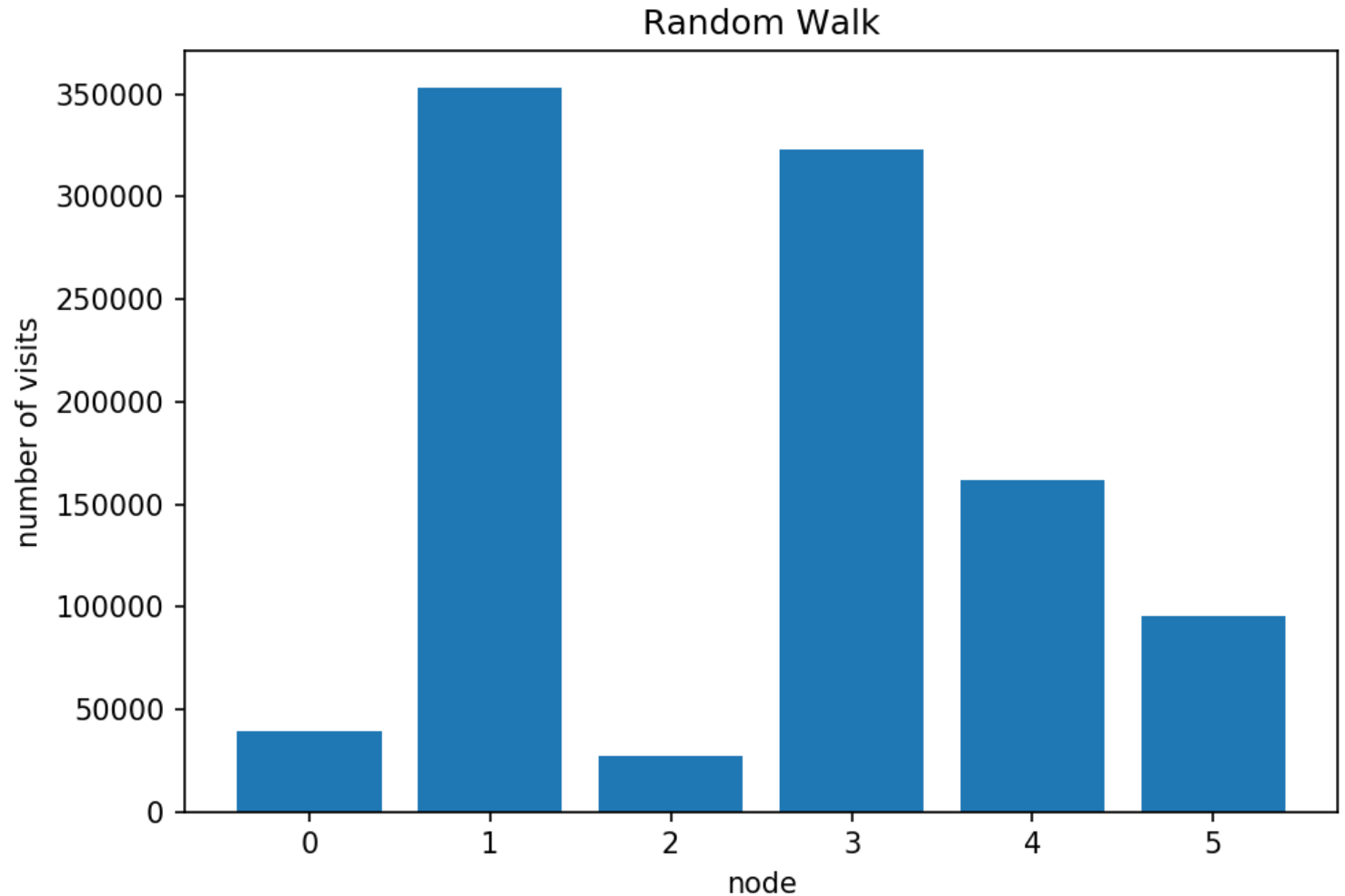
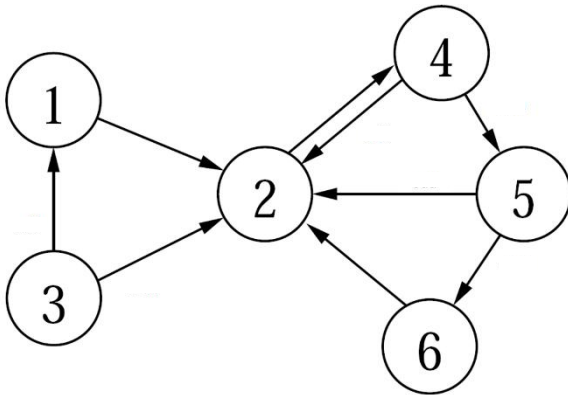
```
| [[1], [3], [0, 1], [1, 4], [1, 5], [1]]
  [0.039365 0.353211 0.02751  0.322593 0.1623  0.095021]
```



Simulate random walk (random surfer model)

pagerank.ipynb

```
import matplotlib.pyplot as plt
plt.bar(range(6), count)
plt.title('Random Walk')
plt.xlabel('node')
plt.ylabel('number of visits')
plt.show()
```



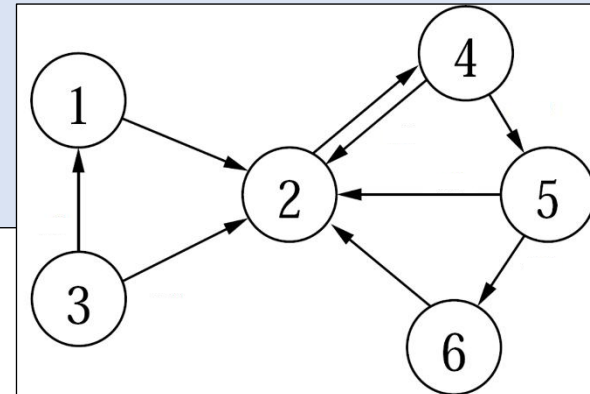
Transition matrix A

pagerank.ipynb

```
A = G / degree # Normalize row sums to one. Note that 'degree'
                # is an n x 1 matrix, whereas G is an n x n matrix.
                # The elementwise division is repeated for each column of G
print(A)
```

Python shell

```
| [[0.  1.  0.  0.  0.  0. ]
   [0.  0.  0.  1.  0.  0. ]
   [0.5 0.5 0.  0.  0.  0. ]
   [0.  0.5 0.  0.  0.5 0. ]
   [0.  0.5 0.  0.  0.  0.5]
   [0.  1.  0.  0.  0.  0. ]]
```



Repeated matrix multiplication

We now want to compute the **probability** $p_j^{(i)}$ to be in vertex j after i steps. Let $p^{(i)} = (p_0^{(i)}, \dots, p_{n-1}^{(i)})$.

Initially we have $p^{(0)} = (1, 0, \dots, 0)$.

We compute a matrix M , such that $p^{(i)} = M^i \cdot p^{(0)}$ (assuming $p^{(0)}$ is a column vector).

If we let $\mathbf{1}_n$ denote the $n \times n$ matrix with 1 in each entry, then M can be computed as:

$$p_j^{(i+1)} = \frac{1}{6} \cdot \frac{1}{n} + \frac{5}{6} \sum_k p_k^{(i)} \cdot A_{k,j}$$

$$p^{(i+1)} = \underbrace{\left(\frac{1}{6} \cdot \frac{1}{n} \mathbf{1}_n + \frac{5}{6} A^T \right)}_M \cdot p^{(i)}$$

pagerank.ipynb

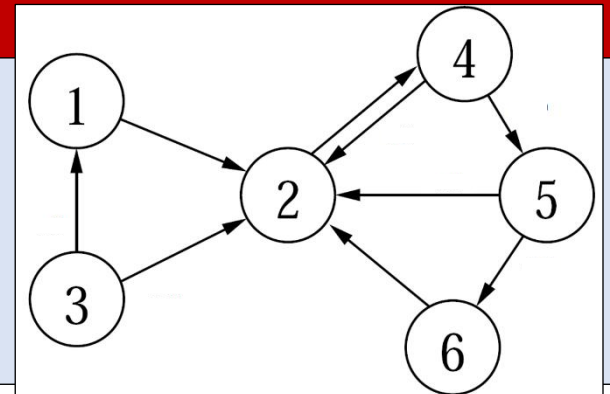
```
ITERATIONS = 20
p_0 = np.zeros((n, 1))
p_0[0, 0] = 1.0
M = 1 / (6 * n) + 5 / 6 * A.T

p = p_0
prob = p # 'prob' will contain each
          # computed 'p' as a new column
for _ in range(ITERATIONS):
    p = M @ p
    prob = np.append(prob, p, axis=1)

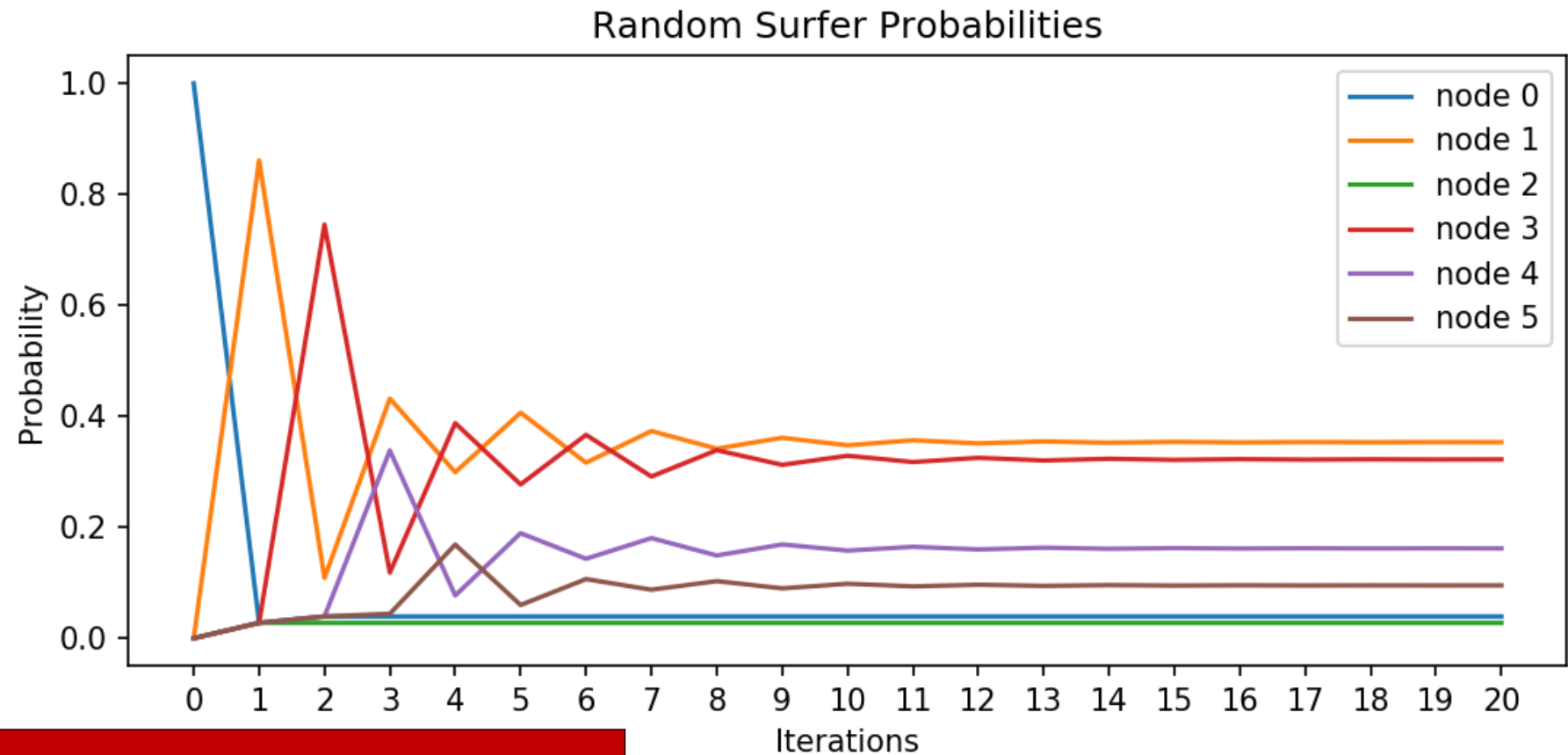
print(p)
```

Python shell

```
| [[0.03935185]
  [0.35326184]
  [0.02777778]
  [0.32230071]
  [0.16198059]
  [0.09532722]]
```



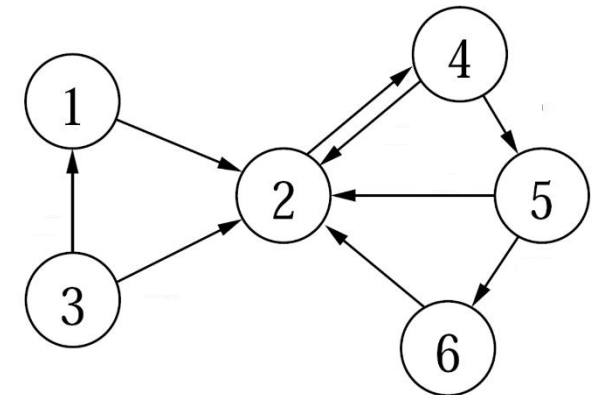
Rate of convergence



pagerank.ipynb

```
x = range(ITERATIONS + 1)
for node in range(n):
    plt.plot(x, prob[node], label=f'node {node}')

plt.xticks(x)
plt.title('Random Surfer Probabilities')
plt.xlabel('Iterations')
plt.ylabel('Probability')
plt.legend()
plt.show()
```



Repeated squaring

$$\underbrace{M \cdot (\cdots (M \cdot (M \cdot p^{(0)})) \cdots)}_{k \text{ multiplications, } k \text{ power of } 2} = M^k \cdot p^{(0)} = M^{2^{\log_2 k}} \cdot p^{(0)} = (\cdots ((M^2)^2)^2 \cdots)^2 \cdot p^{(0)}$$

k multiplications, k power of 2

pagerank.ipynb

```
from math import log2
MP = M
for _ in range(1 + int(log2(ITERATIONS))):
    MP = MP @ MP
p = MP @ p_0
print(p)
```

Python shell

```
| [[0.03935185]
  [0.35332637]
  [0.02777778]
  [0.32221711]
  [0.16203446]
  [0.09529243]]
```

PageRank : Computing eigenvector for $\lambda = 1$

- We want to find a vector p , with $|p| = 1$, where $Mp = p$, i.e. an *eigenvector* p for the eigenvalue $\lambda = 1$

pagerank.ipynb

```
eigenvalues, eigenvectors = np.linalg.eig(M)
idx = eigenvalues.argmax()      # find the largest eigenvalue (= 1)
p = np.real(eigenvectors[:, idx]) # .real returns the real part of complex numbers
p /= p.sum()                    # normalize p to have sum 1
print(p)
```

Python shell

```
| [0.03935185 0.3533267 0.02777778 0.32221669 0.16203473 0.09529225]
```

PageRank : Note on practicality

- In practice an explicit matrix for billions of nodes is infeasible, since the number of entries would be order of 10^{18}
- Instead use **sparse matrices** (in Python modul `scipy.sparse`) and stay with repeated multiplication

Linear programming

scipy.optimize.linprog

- `scipy.optimize.linprog` can solve *linear programs* of the following form, where one wants to find an $n \times 1$ vector x satisfying:

Minimize: $c^T \cdot x$

Subject to: $A_{ub} \cdot x \leq b_{ub}$
 $A_{eq} \cdot x = b_{eq}$

dimension

$c : n \times 1$

$A_{ub} : m \times n$

$b_{ub} : m \times 1$

$A_{eq} : k \times n$

$b_{eq} : k \times 1$

Some other open-source optimization libraries [PuLP](#) and [Pyomo](#)
For industrial strength linear solvers, use solvers like [Cplex](#) or [Gurobi](#) (mixed-integer linear programs)

Linear programming example

Maximize

$$3 \cdot x_1 + 2 \cdot x_2$$

Subject to

$$2 \cdot x_1 + 1 \cdot x_2 \leq 10$$

$$5 \cdot x_1 + 6 \cdot x_2 \geq 4$$

$$-3 \cdot x_1 + 7 \cdot x_2 = 8$$



Minimize

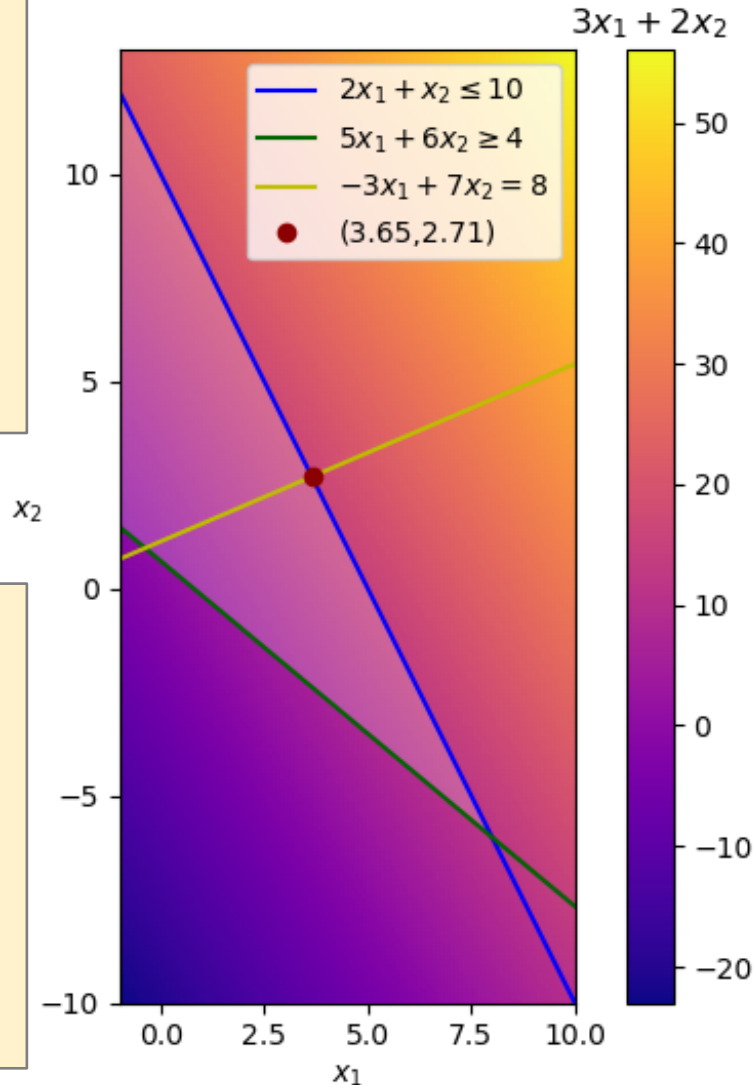
$$-(3 \cdot x_1 + 2 \cdot x_2)$$

Subject to

$$2 \cdot x_1 + 1 \cdot x_2 \leq 10$$

$$-5 \cdot x_1 + -6 \cdot x_2 \leq -4$$

$$-3 \cdot x_1 + 7 \cdot x_2 = 8$$



linear_programming.py

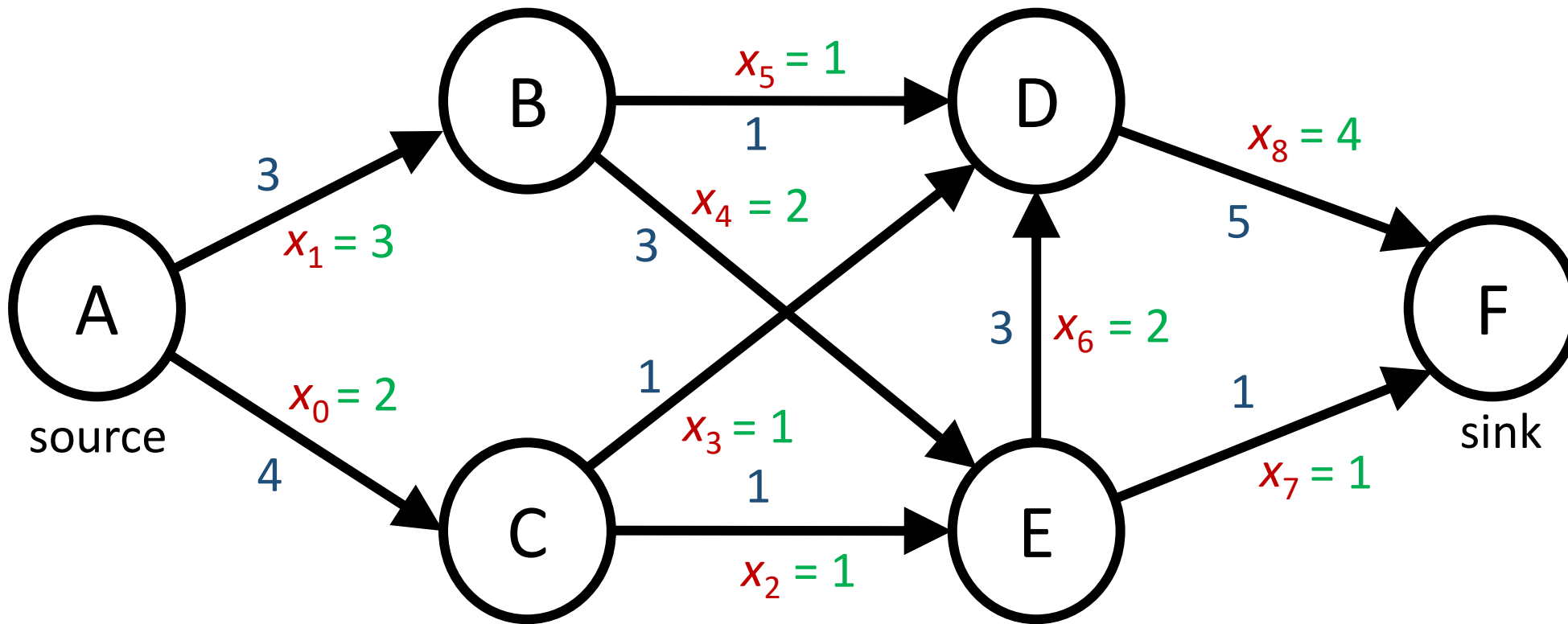
```
import numpy as np
from scipy.optimize import linprog
c = np.array([3, 2])
A_ub = np.array([[ 2,  1],
                 [-5, -6]]) # multiplied by -1
b_ub = np.array([10, -4])
A_eq = np.array([[ -3,  7]])
b_eq = np.array([8])
res = linprog(-c, # maximize = minimize the negated
              A_ub=A_ub,
              b_ub=b_ub,
              A_eq=A_eq,
              b_eq=b_eq)
print(res) # res.x is the optimal vector
```

Python shell

```
fun: -16.35294117647059
message: 'Optimization terminated successfully.'
nit: 3
slack: array([ 0.          , 30.47058824])
status: 0
success: True
x: array([3.64705882, 2.70588235])
```

Maximum flow

Solving maximum flow using linear programming



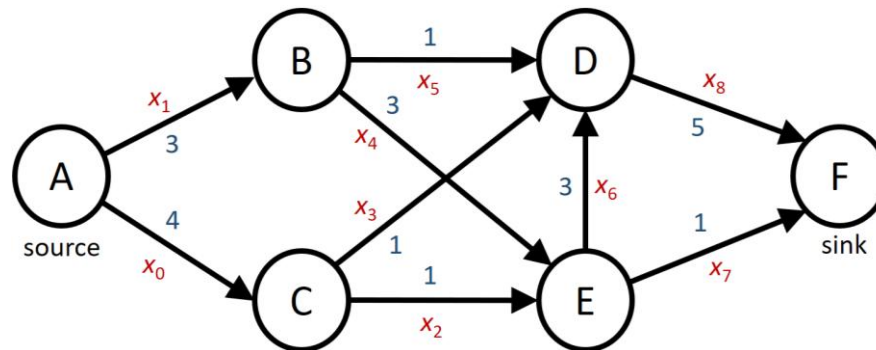
We will use the `scipy.optimize.linprog` function to solve the *maximum flow* problem on the above directed graph. We want to send as much *flow* from node A to node F. Edges are **numbered 0..8** and each edge has a maximum *capacity*.

Maximize	flow value
$x_7 + x_8$	
Subject to	
$x_0 \leq 4$	capacity constraints
$x_1 \leq 3$	
$x_2 \leq 1$	
$x_3 \leq 1$	
$x_4 \leq 3$	
$x_5 \leq 1$	
$x_6 \leq 3$	
$x_7 \leq 1$	
$x_8 \leq 5$	
$x_1 = x_4 + x_5$	flow conservation
$x_0 = x_2 + x_3$	
$x_3 + x_5 + x_6 = x_8$	
$x_2 + x_4 = x_6 + x_7$	

Note: solution not unique

Solving maximum flow using linear programming

- x is a vector describing the flow along each edge
- c is a vector to add the flow along the edges (7 and 8) to the sink (F), i.e. a function computing *the flow value*
- A_{ub} and b_{ub} is a set of *capacity constraints*, for each edge $\text{flow} \leq \text{capacity}$
- A_{eq} and b_{eq} is a set of *flow conservation* constraints, for each non-source and non-sink node (B, C, D, E), requiring that the flow into equals the flow out of a node



Minimize

$-x_7 - x_8$
 $c^T \cdot x$

flow value

Subject to

$x_0 \leq 4$
 $x_1 \leq 3$
 $x_2 \leq 1$
 $x_3 \leq 1$
 $x_4 \leq 3$
 $x_5 \leq 1$
 $x_6 \leq 3$
 $x_7 \leq 1$
 $x_8 \leq 5$

$A_{ub} \cdot x \leq b_{ub}$
 $\Updownarrow$
 $x \leq \text{capacity}$

capacity constraints

$A_{eq} \cdot x = b_{eq} = 0$

$0 = -x_1 + x_4 + x_5$
 $0 = -x_0 + x_2 + x_3$
 $0 = -x_3 - x_5 - x_6 + x_8$
 $0 = -x_2 - x_4 + x_6 + x_7$

flow conservation

maximum-flow.py

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

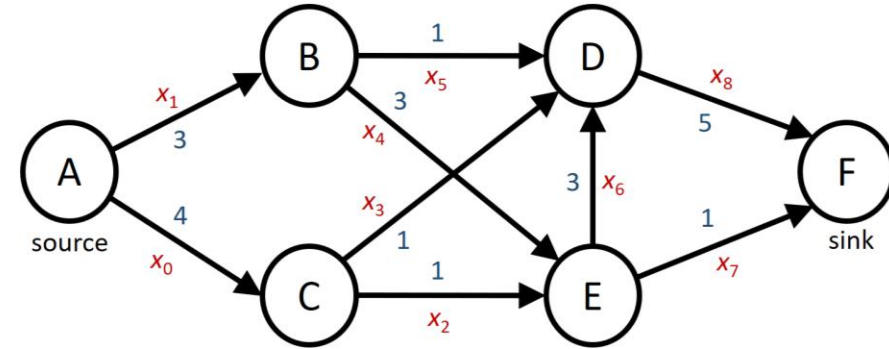
#           0  1  2  3  4  5  6  7  8
conservation = np.array([[ 0,-1, 0, 0, 1, 1, 0, 0, 0], # B
                        [-1, 0, 1, 1, 0, 0, 0, 0, 0], # C
                        [ 0, 0, 0,-1, 0,-1,-1, 0, 1], # D
                        [ 0, 0,-1, 0,-1, 0, 1, 1, 0]]) # E

#           0  1  2  3  4  5  6  7  8
sinks = np.array([0, 0, 0, 0, 0, 0, 0, 1, 1])

#           0  1  2  3  4  5  6  7  8
capacity = np.array([4, 3, 1, 1, 3, 1, 3, 1, 5])

res = linprog(-sinks,
              A_eq=conservation,
              b_eq=np.zeros(conservation.shape[0]),
              A_ub=np.eye(capacity.size),
              b_ub=capacity)

print(res)
```



Python shell

```
| fun: -5.0
| message: 'Optimization terminated successfully.'
| nit: 9
| slack: array([2., 0., 0., 0., 1., 0., 1., 0., 1.])
| status: 0
| success: True
| x: array([2., 3., 1., 1., 2., 1., 2., 1., 4.])
```



the solution found varies
with the scipy version

Generators, iterators

- `__iter__`, `__next__`
- `yield`
- generator expression
- measuring memory usage

Iterable & Iterator

Python shell

```
> L = ['a', 'b', 'c']
> type(L)
| <class 'list'>
> it = L.__iter__()
> type(it)
| <class 'list_iterator'>
> it.__next__()
| 'a'
> it.__next__()
| 'b'
> it.__next__()
| 'c'
> it.__next__()
| StopIteration # Exception
```

Python shell

```
> L = ['a', 'b', 'c']
> it = iter(L) # calls L.__iter__()
> next(it)    # calls it.__next__()
| 'a'
> next(it)
| 'b'
> next(it)
| 'c'
> next(it)
| StopIteration
```

iterator ≈ pointer into list
↓
['a', 'b', 'c']

- Lists are **iterable** (must support `__iter__`)
- `iter` returns an **iterator** (must support `__next__`)

Some iterables in Python: string, list, set, tuple, dict, range, enumerate, zip, map, reversed

Iterator

- `next(iterator_object)` returns the next element from the iterator, by calling the `iterator_object.__next__()`. If no more elements to report, raises exception `StopIteration`
- `next(iterator_object, default)` returns `default` when no more elements are available (no exception is raised)
- for-loops and list comprehensions require iterable objects
`for x in range(5): and [2**x for x in range(5)]`
- The iterator concept is also central to Java and C++

for loop

Python shell

```
> for x in ['a', 'b', 'c']:
    print(x)
```

| a
| b
| c

result of next
on iterator

iterable object
(can call iter on it to
generate an iterator)

=

Python shell

```
> L = ['a', 'b', 'c']
> it = iter(L)
> while True:
    try:
        x = next(it)
    except StopIteration:
        break
    print(x)
```

| a
| b
| c

8.3. The `for` statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt ::= "for" target_list "in" expression_list ":" suite
           ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see [Assignment statements](#)), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), the suite in the `else` clause, if present, is executed, and the loop terminates.

for loop over changing iterable



Changing (extending) the list while scanning
The iterator over a list is just an index into the list

Python shell

```
> L = [1, 2]
> for x in L:
    print(x, L)
    L.append(x + 2)
| 1 [1, 2]
| 2 [1, 2, 3]
| 3 [1, 2, 3, 4]
| 4 [1, 2, 3, 4, 5]
| 5 [1, 2, 3, 4, 5, 6]
...

```

Python shell

```
> L = [1, 2]
> for x in L:
    print(x, L)
    L[:0] = [L[0] - 2, L[0] - 1]
| 1 [1, 2]
| 0 [-1, 0, 1, 2]
| -1 [-3, -2, -1, 0, 1, 2]
| -2 [-5, -4, -3, -2, -1, 0, 1, 2]
| -3 [-7, -6, -5, -4, -3, -2, -1, 0, 1, 2]
...

```

range

Python shell

```
> r = range(1, 6) # 1,2,3,4,5
> type(r)
| <class 'range'>
> it = iter(r)
> type(it)
| <class 'range_iterator'>
> next(it)
| 1
> next(it)
| 2
> for x in it:
    print(x)
| 3
| 4
| 5
> list(r)
| [1, 2, 3, 4, 5]
```

iterable expected
but got iterator ?

create list from iterable

Python shell

```
> it
| <range_iterator object at 0x03E7FFC8>
> iter(it)
| <range_iterator object at 0x03E7FFC8>
> it is iter(it)
| True
```

Calling `iter` on a `range_iterator` just returns the iterator itself, i.e. can use the iterator wherever an iterable is expected

str

Python shell

```
> s = 'abcde'
> list(s) # create list from iterable
| ['a', 'b', 'c', 'd', 'e']
> type(s)
| <class 'str'>
> it = iter(s)
> type(it)
| <class 'str_ascii_iterator'>
> next(it)
| 'a'
> next(it)
| 'b'
> list(it) # iter(it) is it
| ['c', 'd', 'e']
```

Creating an iterable class

names.py

```
class Names:
    def __init__(self, *arg):
        self.people = arg

    def __iter__(self):
        return Names_iterator(self)

class Names_iterator:
    def __init__(self, names):
        self.idx = 0
        self.names = names

    def __next__(self):
        if self.idx >= len(self.names.people):
            raise StopIteration
        self.idx += 1
        return self.names.people[self.idx - 1]

duckburg = Names('Donald', 'Goofy', 'Mickey', 'Minnie')
for name in duckburg:
    print(name)
```

Python shell

```
| Donald
| Goofy
| Mickey
| Minnie
```

class Names
__init__
__iter__

object duckburg
__init__
__class__
people: ('Donald',...)

class Names_iterator
__init__
__next__

object (iterator)
idx: 0
names:
__class__



An infinite iterable

infinite_range.py

```
class infinite_range:
    def __init__(self, start=0, step=1):
        self.start = start
        self.step = step

    def __iter__(self):
        return infinite_range_iterator(self)

class infinite_range_iterator:
    def __init__(self, inf_range):
        self.range = inf_range
        self.current = self.range.start

    def __next__(self):
        value = self.current
        self.current += self.range.step
        return value

    def __iter__(self):  # make iterator iterable
        return self
```

Python shell

```
> r = infinite_range(42, -3)
> it = iter(r)
> for idx, value in zip(range(5), it):
    print(idx, value)
| 0 42
| 1 39
| 2 36
| 3 33
| 4 30
> for idx, value in zip(range(5), it):
    print(idx, value)
| 0 27
| 1 24
| 2 21
| 3 18
| 4 15
> print(sum(r))  # don't do this
| (runs forever)
```



sum and zip take iterables
(zip stops when shortest iterable is exhausted)

Creating an iterable class (iterable = iterator)

my_range.py

```
class my_range:
    def __init__(self, start, end, step):
        self.start = start
        self.end = end
        self.step = step
        self.x = start

    def __iter__(self):
        return self # self also iterator

    def __next__(self):
        if self.x >= self.end:
            raise StopIteration
        answer = self.x
        self.x += self.step
        return answer

r = my_range(1.5, 2.0, 0.1)
```

Python shell

```
> list(r)
| [1.5, 1.6,
  1.7000000000000002,
  1.8000000000000003,
  1.9000000000000004]
> list(r)
| []
```



- Note that object acts both as an iterable and an iterator
- This e.g. also applies to `zip` objects
- Can only iterate over a `my_range` once 

The old sequence iteration protocol

Python shell

```
> class Odd_numbers:
    def __getitem__(self, idx):
        print('getting item', idx)
        if not 0 <= idx < 10:
            raise IndexError
        return 2 * idx + 1

> odds = Odd_numbers()
> odds[3]
> getting item 3
| 7
> it = iter(odds)
> it
| <iterator object at ...>
> print(next(it), next(it), next(it))
| getting item 0
| getting item 1
| getting item 2
| 1 3 5
```

```
> 5 in odds
| getting item 0
| getting item 1
| getting item 2
| True
> 6 in odds
| getting item 0
| getting item 1
| getting item 2
| getting item 3
| getting item 4
| getting item 5
| getting item 6
| getting item 7
| getting item 8
| getting item 9
| getting item 10
| False
```

odds.__contains__ does not exist

- Class with no `__iter__` method but supporting index lookup with `__getitem__`
- Python automatically creates iterator looking up `obj[0]`, `obj[1]`, `obj[2]`, ... until `IndexError` raised
- Keyword `in` falls back to iteration if no method `__contains__`

itertools

Function

`count(start, step)`
`cycle(seq)`
`repeat(value[, times])`
`chain(seq0, ..., seqk)`
`starmap(func, seq)`
`permutations(seq)`
`islice(seq, start, stop, step)`
...

Description

Infinite sequence: `start, start + step, ...`
Infinite repeats of the elements from `seq`
Infinite repeats of `value` or `times` repeats
Concatenate sequences
`func(*seq[0]), func(*seq[1]), ...`
Generate all possible permutations of `seq`
Create a slice of `seq`
...

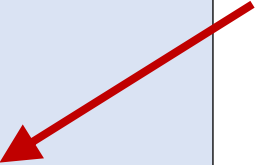
Example : Java iterators

vector-iterator.java

```
import java.util.Vector;
import java.util.Iterator;

class IteratorTest {
    public static void main(String[] args) {
        Vector<Integer> a = new Vector<Integer>();
        a.add(7);
        a.add(42);
        // "C" for-loop & get method
        for (int i=0; i<a.size(); i++)
            System.out.println(a.get(i));
        // iterator
        for (Iterator it = a.iterator(); it.hasNext(); )
            System.out.println(it.next());
        // for-each loop - syntax sugar since Java 5
        for (Integer e : a)
            System.out.println(e);
    }
}
```

In Java iteration does not stop using exceptions, but instead the iterator can be tested if it is at the end of the iterable



Example : C++ iterators

vector-iterator.cpp

```
#include <iostream>
#include <vector>
int main() {
    // Vector is part of STL (Standard Template Library)
    std::vector<int> A = {20, 23, 26};
    // "C" indexing - since C++98
    for (int i = 0; i < A.size(); i++)
        std::cout << A[i] << std::endl;
    // iterator - since C++98
    for (std::vector<int>::iterator it = A.begin(); it != A.end(); ++it)
        std::cout << *it << std::endl;
    // "auto" iterator - since C++11
    for (auto it = A.begin(); it != A.end(); ++it)
        std::cout << *it << std::endl;
    // Range-based for-loop - since C++11
    for (auto e : A)
        std::cout << e << std::endl;
}
```

In C++ iterators can be tested if they reach the end of the iterable



move iterator to next element

Generators

Generator expressions

Python shell

```
> [x ** 2 for x in range(5)] # list comprehension
| [0, 1, 4, 9, 16] # list
> (x ** 2 for x in range(3)) # generator expression
| <generator object <genexpr> at 0x03D9F8A0>
> o = (x ** 2 for x in range(3))
> next(o) # use generator expression as iterator
| 0
> next(o)
| 1
> next(o)
| 4
> next(o)
| StopIteration
```

- A generator expression
(... for x in ...) looks like a list comprehension, except square brackets are replaced by parenthesis
- Is an iterable and iterator, that uses less memory than a list comprehension
- computation is done *lazily*, i.e. first when needed

Nested generator expressions

Python shell

```
> squares = (x ** 2 for x in range(1, 6)) # generator expression
> ratios = (1 / y for y in squares) # generator expression
> ratios
| <generator object <genexpr> at 0x031FC230>
> next(ratios)
| 1.0
> next(ratios)
| 0.25
> list(ratios)
| [0.1111111111111111, 0.0625, 0.04] # remaining 3
```

- Each fraction is first computed when requested by `next(ratios)` (implicitly called repeatedly in `list(ratios)`)
- The next value of `squares` is first computed when needed by `ratios`

Generator expressions as function arguments

Python shell

```
> doubles = (x * 2 for x in range(1, 6))
> sum(doubles)    # sum takes an iterable
| 30
> sum((x * 2 for x in range(1, 6)))
| 30
> sum(x * 2 for x in range(1, 6))    # one pair of parenthesis omitted
| 30
```

- Python allows to omit a pair of parenthesis when a generator expression is the only argument to a function

`f(... for x in ...)` $\equiv$ `f((... for x in ...))`

Generator functions

two.py

```
def two():  
    yield 1  
    yield 2
```

Python shell

```
> two()  
| <generator object two at 0x03629510>  
> t = two()  
> next(t)  
| 1  
> next(t)  
| 2  
> next(t)  
| StopIteration
```

- A *generator function* contains one or more `yield` statements
- Python automatically makes a call to a generator function into an iterable and iterator (provides `__iter__` and `__next__`)
- Calling a generator function returns a *generator object*
- Whenever `next` is called on a generator object, the executing of the function continues until the next `yield exp` and the value of `exp` is returned as a result of `next`
- Reaching the end of the function or a return statement, will raise `StopIteration`
- Once consumed, can't be reused

Generator functions (II)

`my_generator.py`

```
def my_generator(n):  
    yield 'Start'  
    for i in range(n):  
        yield chr(ord('A') + i)  
    yield 'Done'
```

`Python shell`

```
> g = my_generator(3)  
> print(g)  
| <generator object my_generator at 0x03E2F6F0>  
> print(list(g))  
| ['Start', 'A', 'B', 'C', 'Done']  
> print(list(g)) # generator object g exhausted  
| []  
> print(*my_generator(5)) # * takes an iterable (PEP 448)  
| Start A B C D E Done
```

Generator functions (III)

```
my_range_generator.py
```

```
def my_range(start, end, step):  
    x = start  
    while x < end:  
        yield x  
        x += step
```

```
Python shell
```

```
> list(my_range(1.5, 2.0, 0.1))  
| [1.5, 1.6, 1.7000000000000002, 1.8000000000000003, 1.9000000000000004]
```

- Generator functions are often easier to write than creating an iterable class and the accompanying iterator class

Pipelining generators

Python shell

```
> def squares(seq):      # seq should be an iterable object
    for x in seq:        # use iterator to run through seq
        yield x ** 2     # generator

> list(squares(range(5)))
| [0, 1, 4, 9, 16]

> list(squares(squares(range(5))))    # pipelining generators
| [0, 1, 16, 81, 256]

> sum(squares(squares(range(100000000))))    # pipelining generators
| 199999995000000003333333333333333333330000000

> sum((x ** 2) ** 2 for x in range(100000000))    # generator expression
| 199999995000000003333333333333333333330000000

> sum([(x ** 2) ** 2 for x in range(100000000)])    # list comprehension
| MemoryError    # when using a 32-bit version of Python, limited to 2 GB
```

yield vs yield from

Python shell

```
> def g():  
    yield 1  
    yield [2, 3, 4]  
    yield 5  
  
> list(g())  
| [1, [2, 3, 4], 5]
```

Python shell

```
> def g():  
    yield 1  
    yield from [2, 3, 4]  
    yield 5  
  
> list(g())  
| [1, 2, 3, 4, 5]
```

- `yield from` available since Python 3.3
- `yield from exp` $\approx$ `for x in exp: yield x`

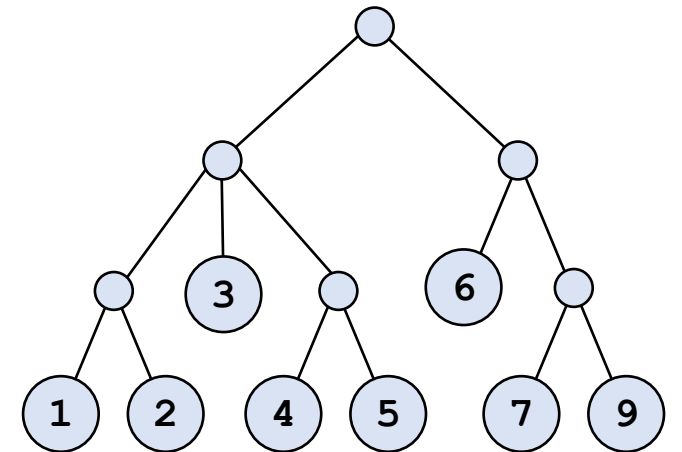
Recursive yield from

without **yield from**

```
for value in traverse(child):  
    yield value
```

Python shell

```
> def traverse(T): # recursive generator  
    if isinstance(T, tuple):  
        for child in T:  
            yield from traverse(child)  
    else:  
        yield T  
  
> T = (((1, 2), 3, (4, 5)), (6, (7, 9)))  
> traverse(T)  
| <generator object traverse at 0x03279F30>  
> list(traverse(T))  
| [1, 2, 3, 4, 5, 6, 7, 9]
```



Generator .close()

random_integers.py

```
def random_integers(n):
    seen = set() # integers generated so far
    while len(seen) < n:
        value = random.randint(1, n)
        if value in seen:
            continue # skip duplicates
        seen.add(value)
        yield value
```

- A generator can be forced to terminate by calling **.close()**
- Technically, the generator continues with a **GeneratorExit** exception raised, so that it can clean up before returning
- Useful to release resources from generators, e.g., infinite generators

Python shell

```
> r = random_integers(10)
> next(r)
| 9
> next(r)
| 3
> next(r)
| 5
> r.close()
> next(r)
| StopIteration

> r = random_integers(1_000_000_000)
> next(r)
| 191907382 # memory usage ~ 10 MB
> sum(next(r) for _ in range(10_000_000))
| 4998986626629771 # memory usage ~ 550 MB
> r.close() # memory usage ~ 10 MB
> next(r)
| StopIteration
```

releases the **seen** set containing 10.000.001 elements for garbage collection

Making objects iterable using `yield`

vector2D.py

```
class vector2D:
    def __init__(self, x_value, y_value):
        self.x = x_value
        self.y = y_value

    def __iter__(self): # generator
        yield self.x
        yield self.y

    def __iter__(self): # alternative generator
        yield from (self.x, self.y)

v = vector2D(5, 7)
print(list(v))
print(tuple(v))
print(set(v))
```

Python shell

```
| [5, 7]
| (5, 7)
| {5, 7}
```


Generators vs iterables

- Iterables can often be reused (like lists, tuples, strings)
- Generators cannot be reused (only if a new generator object is created, starting over again)
- David Beazley's tutorial on
“*Generators: The Final Frontier*”, PyCon 2014 (3:50:54)
Throughout advanced discussion of generators, e.g. how to use `.send`
method to implement coroutines
<https://www.youtube.com/watch?v=D1tw9kLmYg>

Measuring memory usage

Measuring memory usage (memory profiling)

- Macro level:

Task Manager (Windows)
Activity Monitor (Mac)
top (Linux)

- Variable level:

`getsizeof` from `sys` module

- Detailed overview:

Module `memory_profiler`

Allows detailed space usage of the code line-by-line (using `@profile` function decorator) or a plot of total space usage over time

`pip install memory-profiler`

Python shell

```
> import sys
> sys.getsizeof(42)
| 28 # size of the integer 42 is 28 bytes
> sys.getsizeof(42 ** 42)
| 56 # the size increases with value
> sys.getsizeof('42')
| 51 # size of a string
> import numpy as np
> sys.getsizeof(np.array(range(100), dtype='int32'))
| 512 # also works on Numpy arrays
> squares = [x ** 2 for x in range(1000000)]
> sys.getsizeof(squares)
| 8448728
> g = (x ** 2 for x in range(1000000))
> sys.getsizeof(g)
| 208
```



size values depend on the Python version, e.g., 32 vs 64 bit

Module

memory-profiler

pypi.org/project/memory-profiler/

memory_usage.py

```
from memory_profiler import profile

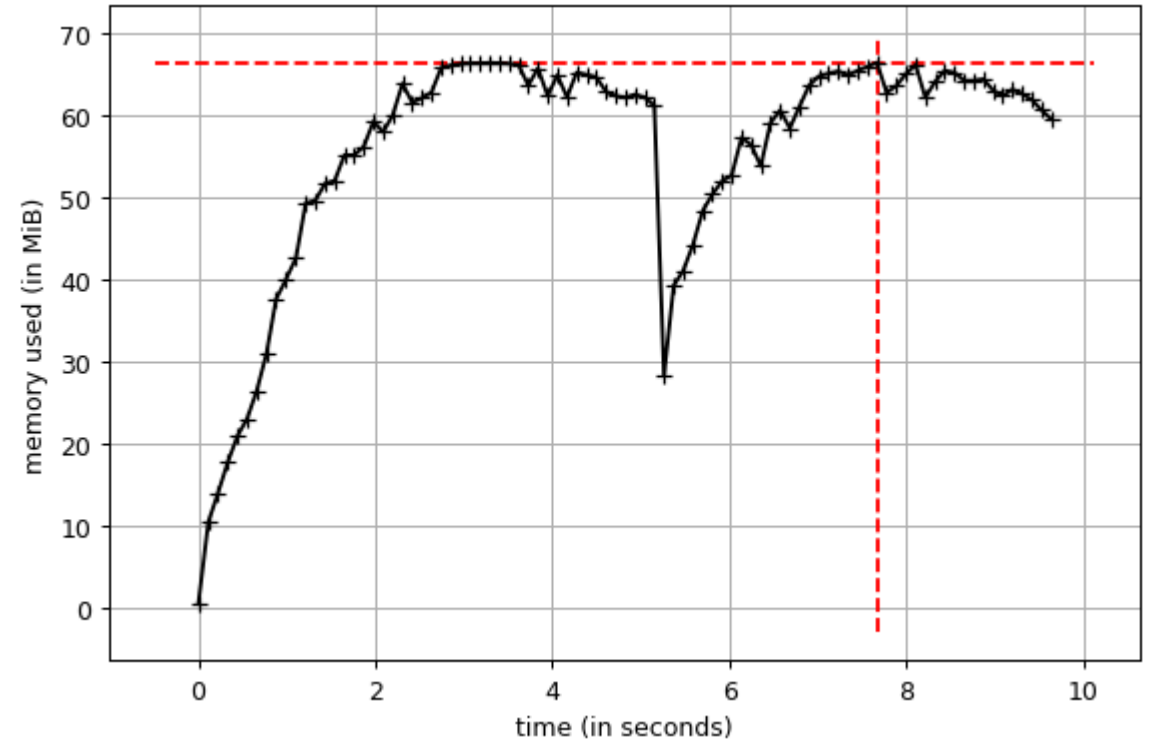
@profile # prints new statistics for each call
def use_memory():
    s = 0
    x = list(range(20_000_000))
    s += sum(x)
    y = list(range(10_000_000))
    s += sum(x)

use_memory()
```

Python Shell

Filename: C:/.../memory_usage.py

Line #	Mem usage	Increment	Line Contents
3	32.0 MiB	32.0 MiB	@profile
4			def use_memory():
5	32.0 MiB	0.0 MiB	s = 0
6	415.9 MiB	383.9 MiB	x = list(range(20_000_000))
7	415.9 MiB	0.0 MiB	s += sum(x)
8	607.8 MiB	191.9 MiB	y = list(range(10_000_000))
9	607.8 MiB	0.0 MiB	s += sum(x)



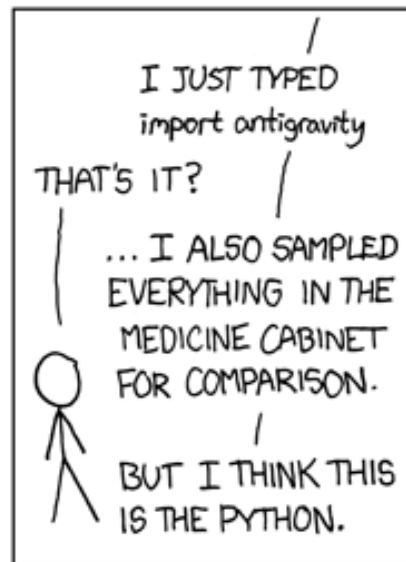
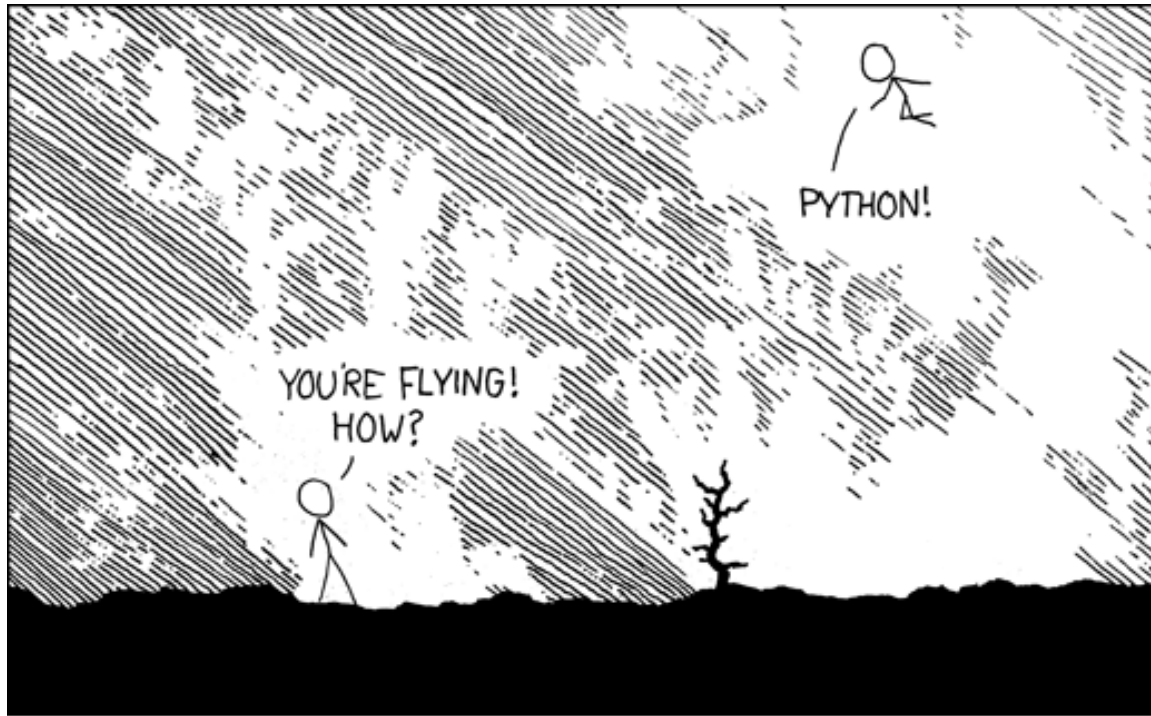
memory_sin_usage.py

```
from math import sin, pi

for a in range(1000):
    x = list(range(int(1000000 * sin(pi * a / 250))))
```

Windows Shell

```
> pip install memory-profiler
> mprof run memory_sin_usage.py
| mprof: Sampling memory every 0.1s
| running as a Python program...
> mprof plot
```



xkcd.com/353

Modules and packages

- import – from – as
- `__name__`, `'__main__'`
- `heapq`

docs.python.org/3/tutorial/modules.html

Python modules and packages

- A Python **module** is a *module_name.py* file containing Python code
- A Python **package** is a collection of modules

Why do you need modules ?

- A way to structure code into **smaller logical units**
- **Encapsulation** of functionality
- **Reuse** of code in different programs
- You can write your **own modules and packages** or use any of the +600.000 existing packages from pypi.org
- **The Python Standard Library** consists of the modules listed on docs.python.org/3/library



Defining and importing a module

mymodule.py

```
'''This is a 'print something' module.'''

from random import choice

print('Running my module')

def print_something(n):
    W = ['Eat', 'Sleep', 'Rave', 'Repeat']
    print(' '.join(choice(W) for _ in range(n)))

def the_name():
    print(f'__name__ = {__name__}')
```

using_mymodule.py

```
import mymodule

mymodule.the_name()
mymodule.print_something(5)

from mymodule import print_something

print_something(5)
```

Python shell

```
| Running my module
| __name__ = "mymodule"
| Eat Sleep Sleep Sleep Rave
| Eat Sleep Rave Repeat Sleep
```

- A module is only run once when imported several times

Some modules mentioned in the course

Module (example functions)	Description
math (pi sqrt ceil log sin)	<i>basic math</i>
random (random randint)	<i>random number generator</i>
numpy (array shape)	<i>multi-dimensional data</i>
pandas	<i>data tables</i>
SQLite	<i>SQL database</i>
scipy scipy.optimize (minimize linprog) scipy.spatial (ConvexHull)	<i>mathematical optimization</i>
matplotlib matplotlib.pyplot (plot show style) matplotlib.backends.backend_pdf (PdfPages) mpl_toolkits.mplot3d (Axes3D)	<i>plotting data</i> <i>print plots to PDF</i> <i>3D plot tools</i>
doctest (testmod) unittest (assertEqual assertTrue)	<i>testing using doc strings</i> <i>unit testing</i>
time (time) datetime (date.today)	<i>current time, conversion of time values</i>
timeit (timeit)	<i>time execution of simple code</i>
heapq	<i>use a list as a heap</i>

Module (example functions)	Description
functools (cache lru_cache total_ordering)	<i>higher order functions and decorators</i>
itertools (islice permutations)	<i>Iterator tools</i>
collections (Counter deque)	<i>data structures for collections</i>
builtins	<i>module containing the Python builtins</i>
os (path)	<i>operating system interface</i>
sys (argv path)	<i>system specific functions</i>
Tkinter PyQt	<i>graphic user interface</i>
xml	<i>xml files (eXtensible Markup Language)</i>
json	<i>JSON (JavaScript Object Notation) files</i>
csv	<i>comma separated files</i>
openpyxl	<i>EXCEL files</i>
re	<i>regular expression, string searching</i>
string (split join lower ascii_letters digits)	<i>string functions</i>

Ways of importing modules

import.py

```
# Import a module name in the current namespace
# All definitions in the module are available as <module>.<name>

import math
print(math.sqrt(2))

# Import only one or more specific definitions into current namespace

from math import sqrt, log, ceil
print(ceil(log(sqrt(100), 2)))

# Import specific modules/definitions from a module into current namespace under new names

from math import sqrt as kvadratrod, \
                    log as logaritme # long import line broken onto multiple lines
import matplotlib.pyplot as plt
print(logaritme(kvadratrod(100)))

# Import all definitions from a module in current namespace
# Deprecated, since unclear what happens to the namespace

from math import *
print(pi) # where did 'pi' come from?
```



Python shell

```
| 1.4142135623730951
| 4
| 2.302585092994046
| 3.141592653589793
```

__all__ vs import *

- A module can control what is imported by `import *` by defining `__all__`

Python shell

```
> min
| <built-in function min>
> sum
| <built-in function sum>
> import numpy
> numpy.min
| <function min at 0x0000024E5DAC71F0>
> numpy.sum
| <function sum at 0x0000024E5DAC5D70>
> from numpy import *
> min
| <function min at 0x0000024E5DAC71F0>
> sum
| <function sum at 0x0000024E5DAC5D70>
> numpy.__all__
| [..., 'min', ..., 'sum', ...] # len(numpy.__all__) == 500
```

all.py

```
__all__ = ['f']
def f():
    print('this is f')
def g():
    print('this is g')
```

Python shell

```
> import all
> all.f()
| this is f
> all.g()
| this is g
> from all import *
> f()
| this is f
> g()
| NameError: name 'g' is not defined
```

Performance of different ways of importing

`from math import sqrt`

appears to be faster than

`math.sqrt`

`sqrt_performance.py`

```
from time import time

import math
start = time()
x = sum(math.sqrt(x) for x in range(10000000))
end = time()
print('math.sqrt', end - start)

from math import sqrt
start = time()
x = sum(sqrt(x) for x in range(10000000))
end = time()
print('from math import sqrt', end - start)

def test(sqrt=math.sqrt): # abuse of keyword argument
    start = time()
    x = sum(sqrt(x) for x in range(10000000))
    end = time()
    print('bind sqrt to keyword argument', end - start)

test()
```

Python shell

```
| math.sqrt 4.05187726020813
| from math import sqrt 3.5011463165283203
| bind sqrt to keyword argument 3.261594772338867
```

Listing definitions in a module: `dir(module)`

Python shell

```
> import math
> import matplotlib.pyplot as plt
> dir(math)
| ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
| 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
| 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
| 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
| 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
| 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
| 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
> help(math)
| Help on built-in module math:
| NAME
|     math
| DESCRIPTION
|     ...
```

name

double.py

```
'''Module double.'''

def f(x):
    '''
    Some doc test code:

    >>> f(21)
    42
    >>> f(7)
    14
    '''

    return 2 * x

print(f'{{__name__}} = {{}}')

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Python shell

```
| __name__ = __main__
...
2 passed and 0 failed.
Test passed.
```

using_double.py

```
import double

print(f'{{__name__}} = {{}}')
print(double.f(5))
```

Python shell

```
| __name__ = 'double'
__name__ = '__main__'
10
```

- The variable `__name__` contains the name of the module, or `'__main__'` if the file is run as the main file by the interpreter
- Can e.g. be used to test a module if the module is run independently

module importlib

- Implements the `import` statement (Python internal implementation details)
- `importlib.reload(module)`
 - Reloads a previously imported *module*. Relevant if you have edited the code for the module and want to load the new version in the Python interpreter, without restarting the full program from scratch.



`a_constant.py`

```
the_constant = 7
```

Python shell

```
> import a_constant # import module
> a_constant.the_constant
| 7
> from a_constant import the_constant
> the_constant
| 7
# Update 7 to 42 in a_constant.py
> a_constant.the_constant # new value not reflected
| 7
> import a_constant # void, module already loaded
> a_constant.the_constant
| 7 # unchanged
> import importlib
> importlib.reload(a_constant)
| <module 'a_constant' from 'C:\\...\\a_constant.py'>
> a_constant.the_constant
| 42
> the_constant
| 7 # imported attributes are not updated by reload
> from a_constant import the_constant # force update
> the_constant
| 42 # the new value
```

Packages

- A package is a collection of modules (and subpackages) in a folder = package name
- Only folders having an `__init__.py` file are considered packages
- The `__init__.py` can be empty, or contain code that will be loaded when the package is imported, e.g. importing specific modules

```
mypackage/__init__.py
```

```
mypackage/a.py
```

```
print('Loading mypackage.a')  
def f():  
    print('mypackage.a.f')
```

```
using_mypackage.py
```

```
import mypackage.a  
mypackage.a.f()
```

```
Python shell
```

```
| Loading mypackage.a  
| mypackage.a.f
```

A package with a subpackage

mypackage/__init__.py

```
print('loading mypackage')
```

mypackage/a.py

```
print('Loading mypackage.a')
```

```
def f():  
    print('mypackage.a.f')
```

mypackage/mysubpackage/__init__.py

```
print('loading mypackage.mysubpackage')  
import mypackage.mysubpackage.b
```

mypackage/mysubpackage/b.py

```
print('Loading mypackage.mysubpackage.b')  
def g():  
    print('mypackage.mysubpackage.b.g')
```

using_mysubpackage.py

```
import mypackage.a  
mypackage.a.f()  
import mypackage.mysubpackage  
mypackage.mysubpackage.b.g()  
from mypackage.mysubpackage.b import g  
g()
```

Python shell

```
| loading mypackage  
| Loading mypackage.a  
| mypackage.a.f  
| loading mypackage.mysubpackage  
| Loading mypackage.mysubpackage.b  
| mypackage.mysubpackage.b.g  
| mypackage.mysubpackage.b.g
```


`__pycache__` folder

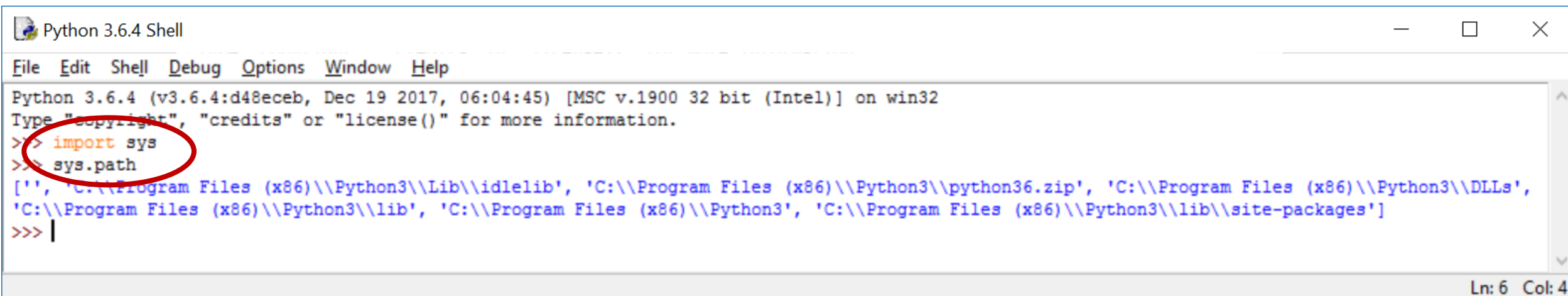
- When Python loads a module the first time it is *compiled* to some intermediate code, and stored as a `.pyc` file in the `__pycache__` folder.
- If a `.pyc` file exists for a module, and the `.pyc` file is newer than the `.py` file, then `import` loads `.pyc` – *saving time* to load the module (but does not make the program itself faster)
- It is safe to delete the `__pycache__` folder – but it will be created again next time a module is loaded

Path to modules

Python searches the following folders for a module in the following order:

- 1) The directory containing the input script / current directory
- 2) *Environment* variable `PYTHONPATH`
- 3) Installation defaults

The function `path` in the modul `sys` returns a list of the paths

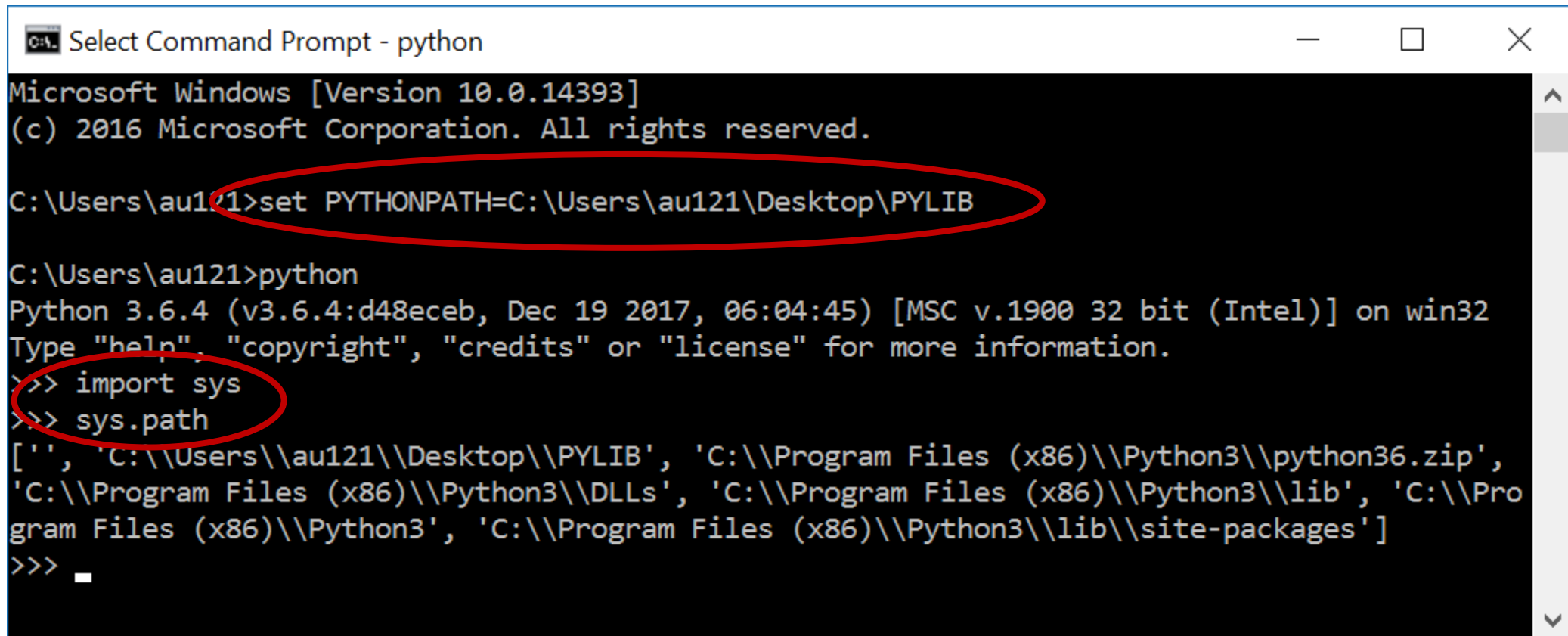


```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> sys.path
['', 'C:\\Program Files (x86)\\Python3\\Lib\\idlelib', 'C:\\Program Files (x86)\\Python3\\python36.zip', 'C:\\Program Files (x86)\\Python3\\DLLs',
'C:\\Program Files (x86)\\Python3\\lib', 'C:\\Program Files (x86)\\Python3', 'C:\\Program Files (x86)\\Python3\\lib\\site-packages']
>>> |
```

Ln: 6 Col: 4

Setting PYTHONPATH from windows shell

- set PYTHONPATH=*paths separated by semicolon*
(only valid until shell is closed)



```
Select Command Prompt - python
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>set PYTHONPATH=C:\Users\au121\Desktop\PYLIB

C:\Users\au121>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', 'C:\\Users\\au121\\Desktop\\PYLIB', 'C:\\Program Files (x86)\\Python3\\python36.zip',
'C:\\Program Files (x86)\\Python3\\DLLs', 'C:\\Program Files (x86)\\Python3\\lib', 'C:\\Pro
gram Files (x86)\\Python3', 'C:\\Program Files (x86)\\Python3\\lib\\site-packages']
>>> _
```

Setting PYTHONPATH from control panel

- Control panel > System > Advanced system settings > Environment Variables > User variables > Edit or New PYTHONPATH

The screenshot illustrates the steps to set the PYTHONPATH environment variable in Windows. The Control Panel window is open, showing the 'System' link in the left sidebar, which is circled in red. The 'System' window is open, showing the 'Advanced' tab, where the 'Environment Variables...' button is circled in red. The 'Environment Variables' dialog box is open, showing the 'User variables for au121' tab. The 'PYTHONPATH' variable is highlighted in blue, and its value is 'C:\Users\au121\Desktop\PYLIB'. The 'New...' and 'Edit...' buttons are circled in red.

Control Panel Home

Device Manager

Remote settings

System protection

Advanced system settings

View basic information

Windows edition

Windows 10 Education

© 2016 Microsoft Corporation. All rights reserved.

System

Processor:

Installed memory (RAM): 12.0 GB (11.8 GB usable)

System Properties

Computer Name Hardware Advanced System Protection Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your sign-in

Settings...

Startup and Recovery

System startup, system failure, and debugging information

Settings...

Environment Variables...

Environment Variables

User variables for au121

Variable	Value
OneDrive	C:\Users\au121\OneDrive
Path	%USERPROFILE%\AppData\Local\Microsoft\WindowsApps\C:\Progr...
PYTHONPATH	C:\Users\au121\Desktop\PYLIB
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

New... Edit... Delete

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
DEFLOGDIR	C:\ProgramData\McAfee\Endpoint Security\Logs
NUMBER_OF_PROCESSORS	4
OS	Windows_NT
Path	C:\Program Files (x86)\Python3\Scripts\;C:\Program Files (x86)\P...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.PY;.PYW
PROCESSOR_ARCHITECTURE	AMD64
PROCESSOR_IDENTIFIER	Intel64 Family 6 Model 78 Stepping 3. GenuineIntel

New... Edit... Delete

OK Cancel Apply

```
> import this
```

```
| The Zen of Python, by Tim Peters
```

```
| Beautiful is better than ugly.
```

```
| Explicit is better than implicit.
```

```
| Simple is better than complex.
```

```
| Complex is better than complicated.
```

```
| Flat is better than nested.
```

```
| Sparse is better than dense.
```

```
| Readability counts.
```

```
| Special cases aren't special enough to break the rules.
```

```
| Although practicality beats purity.
```

```
| Errors should never pass silently.
```

```
| Unless explicitly silenced.
```

```
| In the face of ambiguity, refuse the temptation to guess.
```

```
| There should be one-- and preferably only one --obvious way to do it.
```

```
| Although that way may not be obvious at first unless you're Dutch.
```

```
| Now is better than never.
```

```
| Although never is often better than *right* now.
```

```
| If the implementation is hard to explain, it's a bad idea.
```

```
| If the implementation is easy to explain, it may be a good idea.
```

```
| Namespaces are one honking great idea -- let's do more of those!
```

module `heapq` (Priority Queue)

- Implements a binary **heap** (Williams 1964).
- Stores a set of elements in a standard list, where arbitrary elements can be inserted efficiently and the *smallest element* can be extracted efficiently

`heapq.heappush`
`heapq.heappop`

docs.python.org/3/library/heapq.html

J. W. J. Williams. *Algorithm 232: Heapsort*. Communications of the ACM (1964)

`heap.py`

```
import heapq
from random import random

H = [] # a heap is just a list

for _ in range(10):
    heapq.heappush(H, random())

while True:
    x = heapq.heappop(H)
    print(x)
    heapq.heappush(H, x + random())
```

`Python shell`

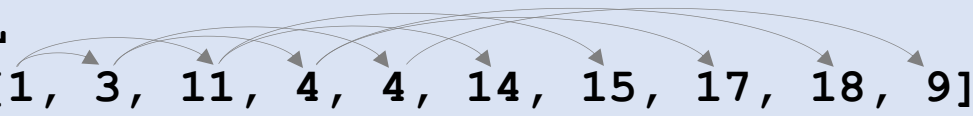
```
| 0.20569933892764458
0.27057819339616174
0.31115615362876237
0.4841062272152259
0.5054280956005357
0.509387117524076
0.598647195480462
0.7035150735555027
0.7073929685826221
0.7091224012815325
0.714213496127318
0.727868481291271
0.8051275413759873
0.8279523767282903
0.8626022363202895
0.9376631236263869
```

Valid heap

- A *valid heap* satisfies for all i :
 $L[i] \leq L[2 \cdot i + 1]$ and $L[i] \leq L[2 \cdot i + 2]$
- **heapify(L)** rearranges the elements in a list to make the list a valid heap

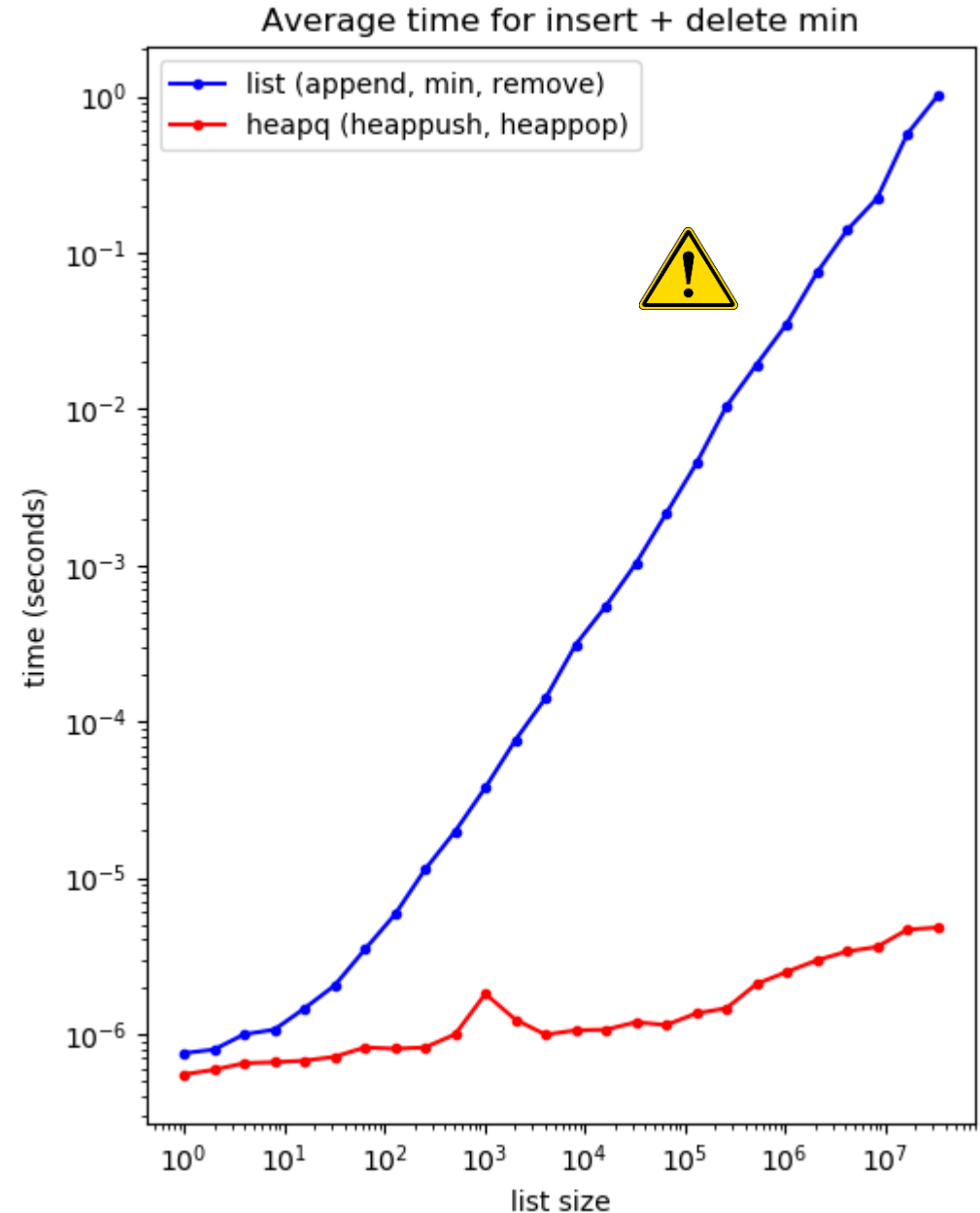
Python shell

```
> from random import randint
> L = [randint(1, 20) for _ in range(10)]
> L # just random numbers
| [18, 1, 15, 17, 4, 14, 11, 3, 4, 9]
> import heapq
> heapq.heapify(L) # make L a valid heap
> L
| [1, 3, 11, 4, 4, 14, 15, 17, 18, 9]
> print(heapq.heappop(L))
| 1
> L
| [3, 4, 11, 4, 9, 14, 15, 17, 18]
> heapq.heappush(L, 7)
> L
| [3, 4, 11, 4, 7, 14, 15, 17, 18, 9]
```



Why heapq ?

- `min` and `remove` on a list take *linear time* (runs through the whole list)
- `heapq` supports `heappush` and `heappop` in *logarithmic time*
- For lists of length 30.000.000 the performance gain is a **factor 200.000**



heap_performance.py (generating plot on previous slide)

```
import heapq
from random import random
import matplotlib.pyplot as plt
from time import time
import gc # garbage collection

size = []
time_heap = []
time_list = []

for i in range(26):
    n = 2 ** i
    size.append(n)

    L = [random() for _ in range(n)]
    R = max(1, 2 ** 23 // n)
    (B) gc.collect()
    start = time()
    for _ in range(R):
        L.append(random())
        x = min(L)
        L.remove(x)
    end = time()
    time_list.append((end - start) / R)
```

```
(A) L = None # avoid MemoryError
    L = [random() for _ in range(n)]
    heapq.heapify(L) # make L a legal heap

(B) gc.collect()
    start = time()
    for _ in range(100000):
        heapq.heappush(L, random())
        x = heapq.heappop(L)
    end = time()
    time_heap.append((end - start) / 100000)

plt.title('Average time for insert + delete min')
plt.xlabel('list size')
plt.ylabel('time (seconds)')
plt.plot(size, time_list, 'b.-',
         label='list (append, min, remove)')
plt.plot(size, time_heap, 'r.-',
         label='heapq (heappush, heappop)')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
```

- (A) Avoid out of memory error for largest experiment, by allowing old `L` to be garbage collected
- (B) Reduce noise in experiments by forcing Python garbage collection before measurement



Working with text

- file formats
- CSV, JSON, XML, Excel
- regular expressions
- module re, finditer

Some file formats

File extension	Content
.html	HyperText Markup Language
.mp3	Audio File
.png .jpeg .jpg	Image files
.svg	Scalable Vector Graphics file
.json	JavaScript Object Notation
.csv	Comma separated values
.xml	eXtensible Markup Language
.xlmx	Micosoft Excel 2010/2007 Workbook

File extension	Description
.exe	Windows executable file
.app	Max OS X Application
.py	Python program
.pyc	Python compiled file
.java	Java program
.cpp	C++ program
.c	C program
.txt	Raw text file

PIL – the Python Imaging Library

- pip install Pillow

```
rotate_image.py  
from PIL import Image  
img = Image.open('Python-Logo.png')  
img_out = img.rotate(45, expand=True)  
img_out.save('Python-rotated.png')
```

- For many file types there exist Python packages handling such files, e.g. for images Pillow supports 40+ different file formats
- For more advanced computer vision tasks you should consider OpenCV



Python-Logo.png



Python-rotated.png

CSV files - Comma Separated Values

- Simple 2D tables are stored as rows in a file, with values separated by comma
- Strings stored are quoted if necessary
- Values read are strings
- The delimiter (default comma) can be changed by keyword argument `delimiter`.
Other typical delimiters are tabs `'\t'`, and semicolon `';'`

csv-example.py

```
import csv
FILE = 'csv-data.csv'
data = [[1, 2, 3],
        ['a', '"b"'],
        [1.0, ['x', 'y'], 'd']]

with open(FILE, 'w', newline='') as outfile:
    csv_out = csv.writer(outfile)
    for row in data:
        csv_out.writerow(row)

with open(FILE, 'r', newline='') as infile:
    for row in csv.reader(infile):
        print(row)
```

Python shell

```
| ['1', '2', '3']
| ['a', '"b"']
| ['1.0', "['x', 'y']", 'd']
```

csv-data.csv

```
1,2,3
a,""b""
1.0,"['x', 'y']",d
```

CSV files - Tab Separated Values

csv-tab-separated.py

```
import csv

FILE = 'tab-separated.csv'

with open(FILE) as infile:
    for row in csv.reader(infile, delimiter='\t'):
        print(row)
```

Python shell

```
| ['1', '2', '3']
  ['4', '5', '6']
  ['7', '8', '9']
```

tab-separated.csv

1	2	3
4	5	6
7	8	9

Reading an Excel generated CSV file

average.py

```
import csv

with open('grades.csv') as file:
    data = csv.reader(file, delimiter=';') # data = iterator over the rows
    header = next(data)                  # ['Name', 'Course', 'Grade']
    count = {}
    total = {}
    for row in data:                      # iterate over data rows
        course = row[header.index('Course')]
        grade = int(row[header.index('Grade')])
        count[course] = count.get(course, 0) + 1
        total[course] = total.get(course, 0) + grade
    print('Average grades:')
    width = max(map(len, count)) # maximum course name length
    for course in count:
        print(f'{course:>{width}s} : {total[course] / count[course]:.2f}')
```

Python shell

```
| Average grades:
|   Analysis : 1.67
| Programming : 1.50
|   Statistics : 2.50
```

	A	B	C
1	Name	Course	Grade
2	Alice	Analysis	1
3	Alice	Programming	1
4	Bob	Statistics	2
5	Alice	Analysis	1
6	Charlie	Analysis	3
7	Charlie	Statistics	3
8	Bob	Programming	2
9			

Saving a file in Excel as
CSV (Comma delimited) (*.csv)

Sometimes use ';' as default separator,
determined by Windows Regional settings

```
File Edit Format View Help
Name;Course;Grade
Alice;Analysis;1
Alice;Programming;1
Bob;Statistics;2
Alice;Analysis;1
Charlie;Analysis;3
Charlie;Statistics;3
Bob;Programming;2
```

CSV files

- Quoting

- The amount of quoting is controlled with keyword argument **quoting**
- `csv.QUOTE_MINIMAL` etc. can be used to select the quoting level
- Depending on choice of quoting, numeric values and strings cannot be distinguished in CSV file (`csv.reader` will read all as strings anyway)

csv-quoting.py

```
import csv
import sys

data = [[1, 1.0, '1.0'], ['abc', '"', '\t"', ',']]

quoting_options = [(csv.QUOTE_MINIMAL, 'QUOTE_MINIMAL'),
                    (csv.QUOTE_ALL, 'QUOTE_ALL'),
                    (csv.QUOTE_NONNUMERIC, 'QUOTE_NONNUMERIC'),
                    (csv.QUOTE_NONE, 'QUOTE_NONE')]

for quoting, name in quoting_options:
    print(name)
    csv_out = csv.writer(sys.stdout, quoting=quoting, escapechar='\\')
    for row in data:
        csv_out.writerow(row)
```

Python shell

```
| QUOTE_MINIMAL # cannot distinguish 1.0 and "1.0"
| 1,1.0,1.0
| abc,"""","" """,",,"
| QUOTE_ALL # cannot distinguish 1.0 and "1.0"
| "1","1.0","1.0"
| "abc","""","" """,",,"
| QUOTE_NONNUMERIC
| 1,1.0,"1.0"
| "abc","""","" """,",,"
| QUOTE_NONE # cannot distinguish 1.0 and "1.0"
| 1,1.0,1.0
| abc,\", \",\,
```


File encodings...

river-utf8.py (size 17 bytes, encoding UTF-8)

Æ Æ U I Æ Å

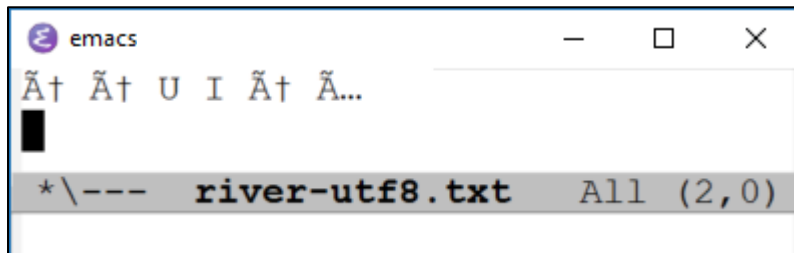


river-windows1252.py (size 13 bytes, encoding Windows-1252)

Æ Æ U I Æ Å



- Text files can be *encoded* using many different encodings (UTF-8, UTF-16, UTF-32, Windows-1252, ANSI, ASCII, ISO-8859-1, ...)
- Different encodings can result in different file sizes, in particular when containing non-ASCII symbols
- Programs often try to predict the encoding of text files (often with success, but not always)
- Opening files assuming wrong encoding can give strange results....



Opening UTF-8 encoded file but trying
to decode using Windows-1252



Opening Windows-1252 encoded file
but trying to decode using UTF-8

encoding.py

river-utf8.py

Æ Æ U I Æ Å

```
for filename in ['river-utf8.txt', 'river-windows1252.txt']:
    print(filename)
    f = open(filename, 'rb')    # open input in binary mode, default = text mode = 't'
    line = f.readline()        # type(line) = bytes = immutable list of integers in 0..255
    print(line)                # byte literals look like strings, prefixed 'b'
    print(list(line))          # print bytes as list of integers
    f = open(filename, 'r', encoding='utf-8') # try to open file as UTF-8
    line = f.readline()        # fails if input line is not utf-8
    print(line)
```

Python shell

```
| river-utf8.txt
| b'\xc3\x86 \xc3\x86 U I \xc3\x86 \xc3\x85\r\n' # \x = hexadecimal value follows
| [195, 134, 32, 195, 134, 32, 85, 32, 73, 32, 195, 134, 32, 195, 133, 13, 10]
| Æ Æ U I Æ Å
|
| river-windows1252.txt
| b'\xc6 \xc6 U I \xc6 \xc5\r\n'
| [198, 32, 198, 32, 85, 32, 73, 32, 198, 32, 197, 13, 10]
| UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc6 in position 0: invalid continuation
byte
> 'Æ Æ U I Æ Å'.encode('utf8') # convert string to (an immutable array of) bytes
| b'\xc3\x86 \xc3\x86 U I \xc3\x86 \xc3\x85'
> 'Æ Æ U I Æ Å'.encode('utf8').decode('Windows-1252') # decode bytes to string
| 'Ã† Ã† U I Ã† Ã...'

```

Reading CSV files with specific encoding

`read_shopping.py`

```
import csv

with open('shopping.csv', encoding='Windows-1252') as file:
    for article, amount in csv.reader(file):
        print('Buy', amount, article)
```

Python shell

```
| Buy 2 æbler
| Buy 4 pærer
| Buy 3 jordbær
| Buy 10 gulerødder
```

`shopping.csv`

```
æbler,2
pærer,4
jordbær,3
gulerødder,10
```

CSV file saved with
Windows-1252 encoding

JSON

*“**JSON** (**J**ava**S**cript **O**bject **N**otation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the **JavaScript** Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is an ideal data-interchange language.”*

www.json.org

- Human readable file format
- Easy way to save a Python expression to a file
- Does *not* support all Python types, e.g. sets are not supported, and tuples are saved (and later loaded) as lists

JSON example

json-example.py

```
import json
FILE = 'json-data.json'
data = ((None, True), (42.7, (42,)), [3, 2, 4], (5, 6, 7),
        {'b': 'banana', 'a': 'apple', 'c': 'coconut'})

with open(FILE, 'w') as outfile:
    json.dump(data, outfile, indent=2, sort_keys=True)

with open(FILE) as infile:
    indata = json.load(infile)

print(indata)
```

Python shell

```
| [[None, True], [42.7, [42]], [3, 2, 4], [5, 6, 7], {'a':
  'apple', 'b': 'banana', 'c': 'coconut'}]
```

json-data.json

```
[
  [
    null,
    true
  ],
  [
    42.7,
    [
      42
    ]
  ],
  [
    3,
    2,
    4
  ],
  [
    5,
    6,
    7
  ],
  {
    "a": "apple",
    "b": "banana",
    "c": "coconut"
  }
]
```

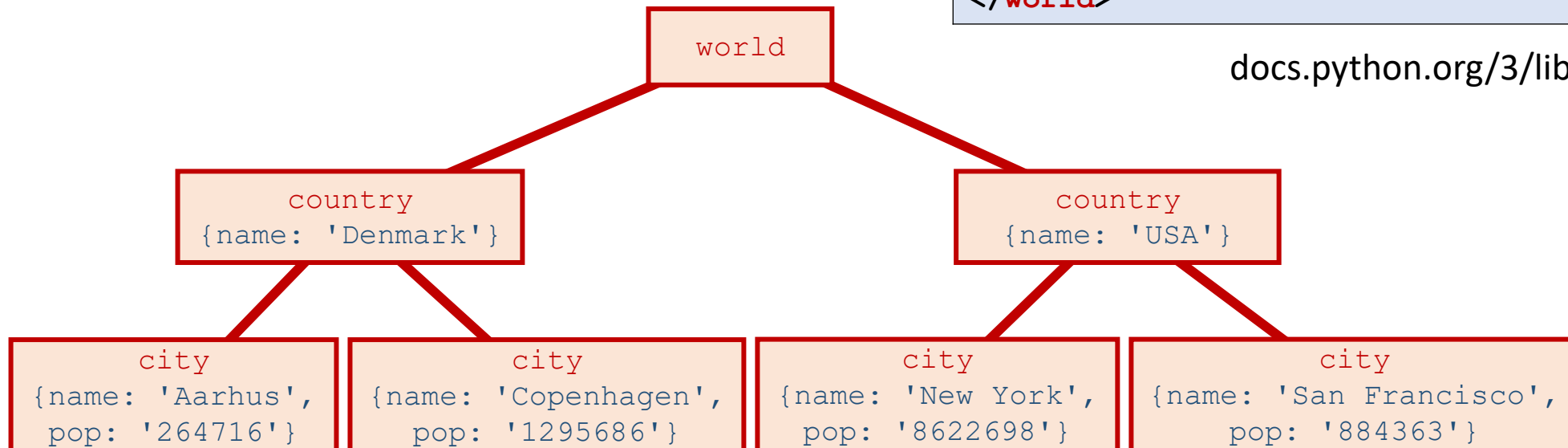
XML - eXtensible Markup Language

- XML is a widespread used data format to store hierarchical data with **tags** and **attributes**

cities.xml

```
<?xml version="1.0"?>
<world>
  <country name="Denmark">
    <city name="Aarhus" pop="264716"/>
    <city name="Copenhagen" pop="1295686"/>
  </country>
  <country name="USA">
    <city name="New York" pop="8622698"/>
    <city name="San Francisco" pop="884363"/>
  </country>
</world>
```

docs.python.org/3/library/xml.html



xml-example.py

```
import xml.etree.ElementTree as ET
FILE = 'cities.xml'
tree = ET.parse(FILE) # parse XML file to internal representation
root = tree.getroot() # get root element
for country in root:
    for city in country:
        print(city.attrib['name'], # get value of attribute for an element
              'in',
              country.attrib['name'],
              'has a population of',
              city.attrib['pop'])
print(root.tag, root[0][1].attrib) # the tag & indexing the children of an element
print([city.attrib['name'] for city in root.iter('city')]) # .iter finds elements
```

Python shell

```
| Aarhus in Denmark has a population of 264716
Copenhagen in Denmark has a population of 1295686
New York in USA has a population of 8622698
San Francisco in USA has a population of 884363
world {'name': 'Copenhagen', 'pop': '1295686'}
['Aarhus', 'Copenhagen', 'New York', 'San Francisco']
```

cities.xml

```
<?xml version="1.0"?>
<world>
  <country name="Denmark">
    <city name="Aarhus" pop="264716"/>
    <city name="Copenhagen" pop="1295686"/>
  </country>
  <country name="USA">
    <city name="New York" pop="8622698"/>
    <city name="San Francisco" pop="884363"/>
  </country>
</world>
```

XML tags with text

city-descriptions.xml

```
<?xml version="1.0"?>
<world>
  <country name="Denmark">
    <city name="Aarhus" pop="264716">The capital of Jutland</city>
    <city name="Copenhagen" pop="1295686">The capital of Denmark</city>
  </country>
  <country name="USA">
    <city name="New York" pop="8622698">Known as Big Apple</city>
    <city name="San Francisco" pop="884363">Home of the Golden Gate Bridge</city>
  </country>
</world>
```

xml-descriptions.py

```
import xml.etree.ElementTree as ET
FILE = 'city-descriptions.xml'
tree = ET.parse(FILE)
root = tree.getroot()

for city in root.iter('city'):
    print(city.get('name'), '-', city.text)
```

Python shell

```
Aarhus - The capital of Jutland
Copenhagen - The capital of Denmark
New York - Known as Big Apple
San Francisco - Home of the Golden Gate Bridge
```


Openpyxl - Microsoft Excel 2010 manipulation

openpyxl-example.py

```
from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill

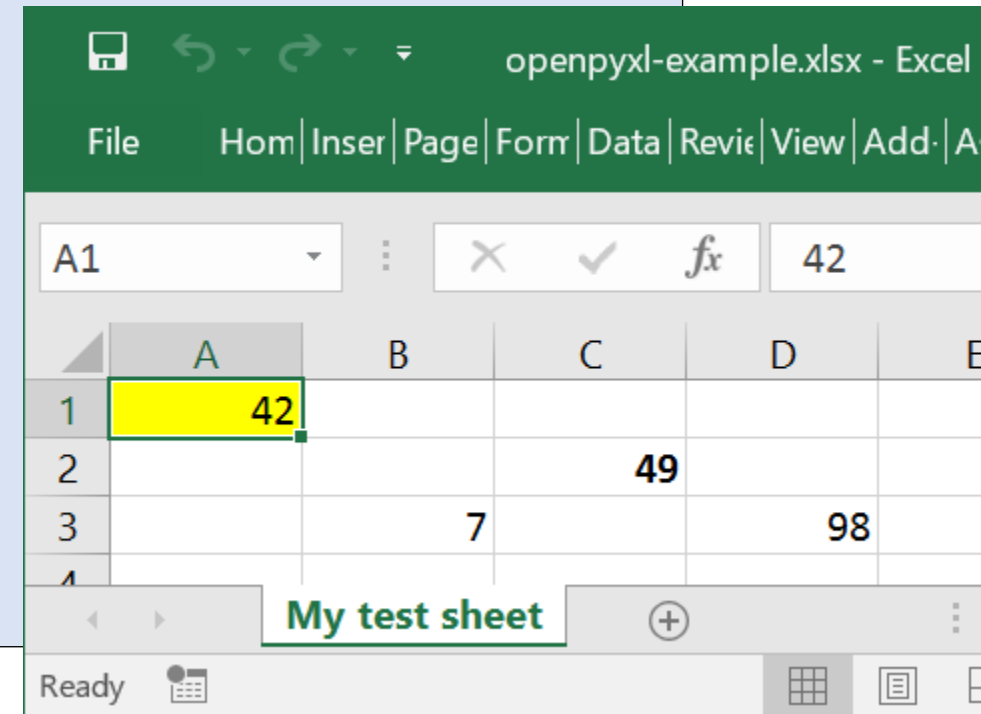
wb = Workbook() # create workbook
ws = wb.active # active worksheet

ws['A1'] = 42
ws['B3'] = 7
ws['C2'] = ws['A1'].value + ws['B3'].value
ws['D3'] = '=A1+B3+C2'

ws.title = 'My test sheet'

ws['A1'].fill = PatternFill('solid', fgColor='ffff00')
ws['C2'].font = Font(bold=True)

wb.save('openpyxl-example.xlsx')
```



String searching using `find`

- Search for first occurrence of *substring* in *str[start, end]*
`str.find(substring[, start[, end]])`
- Returns -1 if no occurrence found.
- `.index` similar as `.find`, except raises `ValueError` exception if substring not found

string-search.py

```
text = 'this is a string - a list of characters'
pattern = 'is'
idx = text.find(pattern)
while idx >= 0:
    print(idx, end=' ')
    idx = text.find(pattern, idx + 1)
```

Python shell

```
| 2 5 22
```

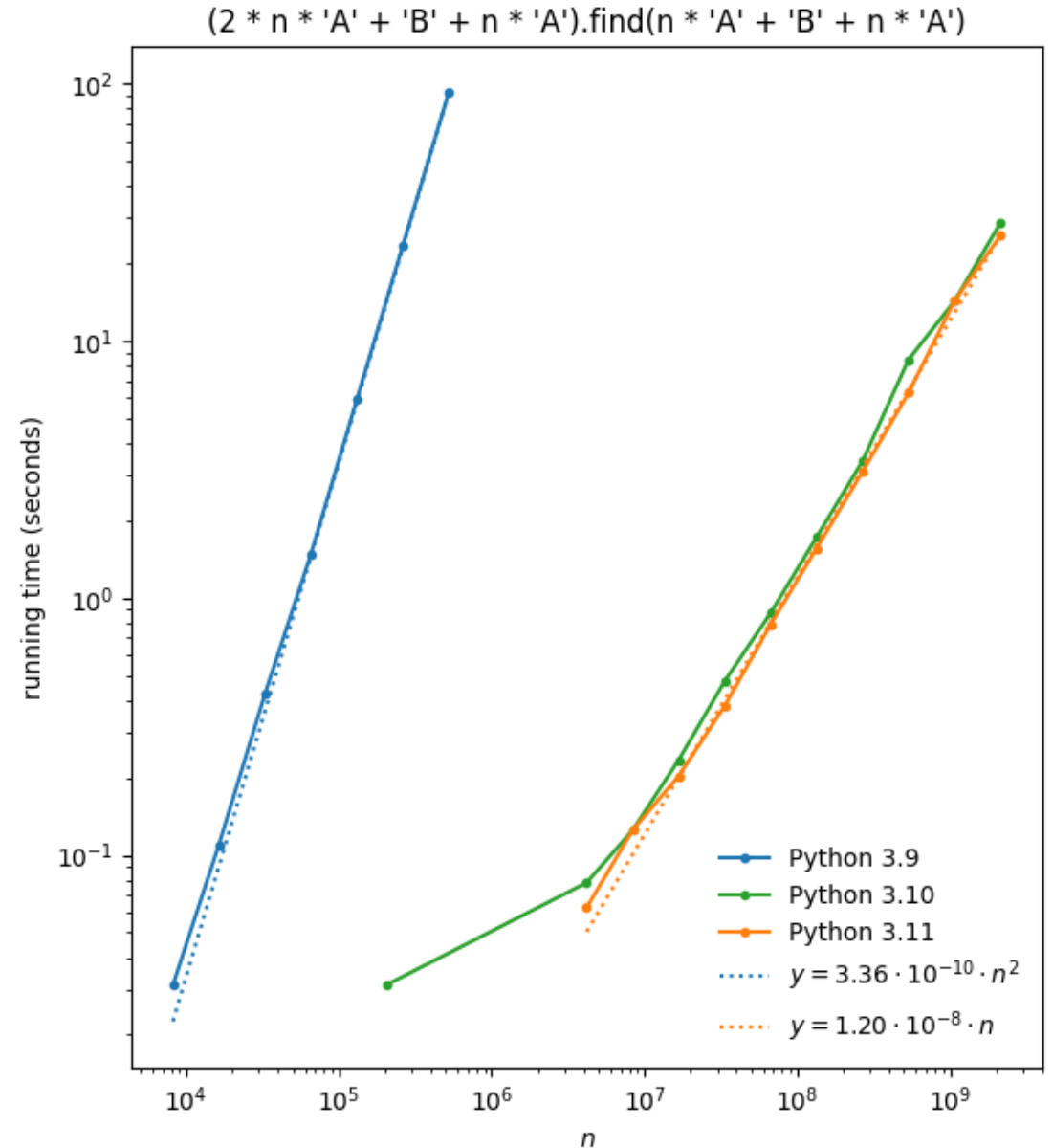
Is *str.find* fast?

- Typically linear
- Until Python 3.9 in some cases quadratic

`"A2nBAn".find("AnBAn")`

- docs.python.org/3/whatsnew/3.10.html

"Substring search functions such as `str1` in `str2` and `str2.find(str1)` now sometimes use Crochemore & Perrin's "Two-Way" string searching algorithm to avoid quadratic behavior on long strings."



Regular expression

– A powerful language to describe sets of strings

■ Examples

- `abc` denotes a string of letters
- `ab*c` any string starting with `a`, followed by an arbitrary number of `b`s and terminated by `c`, i.e. `{ac, abc, abbc, abbbc, abbbbc, ...}`
- `ab+c` equivalent to `abb*c`, i.e. there must be at least one `b`
- `a\wc` any three letter string, starting with `a` and ending with `c`, where second character is any character in `[a-zA-Z0-9_]`
- `a[xyz]c` any three letter string, starting with `a` and ending with `c`, where second character is either `x`, `y` or `z`
- `a[^xyz]c` any three letter string, starting with `a` and ending with `c`, where second character is *none* of `x`, `y` or `z`
- `^xyz` match at start of string (prefix)
- `xyz$` match at end of string (suffix)
- ...

■ See docs.python.org/3/library/re.html for more

String searching using regular expressions

- `re.search(pattern, text)`
 - find the first occurrence of *pattern* in *text* – returns `None` or a *match object*
- `re.findall(pattern, text)`
 - returns a list of non-overlapping occurrences of *pattern* in *text* – returns a list of substrings
- `re.finditer(pattern, text)`
 - iterator returning a match object for each non-overlapping occurrence of *pattern* in *text*

Python shell

```
> import re
> text = 'this is a string - a list of characters'
> re.findall(r'i\w*', text) # prefix with 'r' for raw string literal
| ['is', 'is', 'ing', 'ist']
> for m in re.finditer(r'a[^\t]*t', text): # m is a match object
    print(f'text[{m.start()}, {m.end()}] = {m.group()}')
| text[8, 12] = a st
  text[19, 25] = a list
  text[33, 36] = act
```

Substitution and splitting using regular expressions

- `re.sub(pattern, replacement, text)`
 - replace any occurrence of the *pattern* in *text* by *replacement*
- `re.split(pattern, text)`
 - split *text* at all occurrences of *pattern*

Python shell

```
> text = 'this is a string - a list of characters'
> re.sub(r'\w*i\w*', 'X', text) # replace all words containing i by 'X'
| 'X X a X - a X of characters'
> re.sub(r'\w*i\w*', lambda m: m.group()[::-1], text) # reverse words containing i
| 'siht si a gnirts - a tsil of characters'
> re.split(r'[^\\w]+a[^\\w]+', text) # split around word 'a'
| ['this is', 'string', 'list of characters']
```

Regular expression substitution: \b \w \1 \2 ...

- Assume we want to replace "a" with "an" in front of words starting with the vowels a, e, i, o and u.

Python shell

```
> txt = 'A elephant, a zebra and a ape'      # two places to correct
> re.sub('a', 'an', txt)                      # same as txt.replace('a', 'an')
| 'A elephannt, an zebran annd an anpe'       # replaces all letters 'a' with 'an'
> re.sub(r'\ba\b', 'an', txt)                 # raw string + \b boundary of word
| 'A elephant, an zebra and an ape'           # all lower 'a' replaced
> re.sub(r'\b[aA]\b', 'an', txt)               # both 'a' and 'A' replaced by 'an'
| 'an elephant, an zebra and an ape'          # use () and \1 to reinsert match
> re.sub(r'\b([aA])\b', r'\1n', txt)          # kept 'a' and 'A'
| 'An elephant, an zebra and an ape'
> re.sub(r'\b([aA])\s+[aeiou]', r'\1n', txt)   # \s+ = one or more whitespace
| 'Anlephant, a zebra and anpe'               # missing original whitespace + vowel
> re.sub(r'\b([aA])(\s+[aeiou])', r'\1n\2', txt) # reinsert both () using \1 \2
| 'An elephant, a zebra and an ape'
```

Fun with strings: Lindenmayer systems (L-systems)

Axiom	A	Rules
(1)	AB	$A \rightarrow AB$
(2)	ABA	$B \rightarrow A$
(3)	ABAAB	
(4)	ABAABABA	

First four iterations of parallel rewriting

“L-systems were introduced and developed in 1968 by Aristid Lindenmayer, a Hungarian theoretical biologist and botanist at the University of Utrecht. Lindenmayer used L-systems to describe the behaviour of plant cells and to model the growth processes of plant development.”

- Wikipedia

L_system.py

```
S = 'A'    # axiom

rules = {'A': 'AB', 'B': 'A'}

for i in range(8):
    S = ''.join(rules.get(c, c) for c in S)
    print(S)
```

Python shell

AB

ABA

ABAAB

ABAABABA

ABAABABAABAAB

ABAABABAABAABABAABABA

ABAABABAABAABABAABAABABAABAABAABAAB

A large, stylized blue snowflake graphic composed of many small, interconnected geometric shapes, resembling a cross or a star with multiple points. The snowflake is centered on the page and has a complex, fractal-like structure with many sharp points and intricate internal patterns. It is rendered in a solid blue color against a white background.

```
import matplotlib.pyplot as plt
from math import sin, cos, radians

axiom = 'FX'
rules = {'X': 'X+YF+', 'Y': '-FX-Y'}

def apply_rules(axiom, rules, repeat):
    for _ in range(repeat):
        axiom = ''.join(rules.get(symbol, symbol) for symbol in axiom)
    return axiom

def walk(commands, position=(0, 0), angle=0, turn=90):
    path = [position]
    for move in commands:
        match move:
            case 'F':
                position = (position[0] + cos(radians(angle)),
                           position[1] + sin(radians(angle)))
                path.append(position)
            case '-': angle -= turn
            case '+': angle += turn
    return path

path = walk(apply_rules(axiom, rules, 13))
plt.plot(*zip(*path), '-')
plt.title('Heighway dragon')
plt.show()
```

[illegible]

More space filling curves...

Sierpinski triangle



Axiom F-G-G
 $F \rightarrow F-G+F+G-F$
 $G \rightarrow GG$

Forward F and G
 Turns 120°

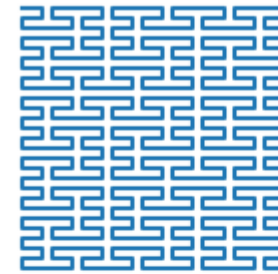
Sierpinski arrowhead curve



Axiom A
 $A \rightarrow B-A-B$
 $B \rightarrow A+B+A$

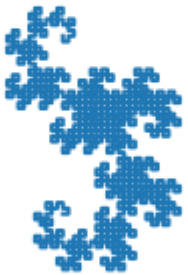
Forward A and B
 Turns 60°

Peano curve



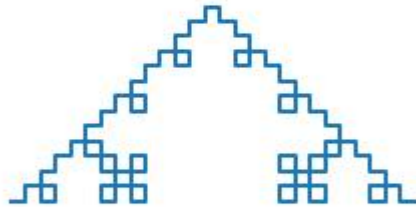
Axiom L
 $L \rightarrow LFRFL-F-RFLFR+F+LFRFL$
 $R \rightarrow RFLFR+F+LFRFL-F-RFLFR$

Heighway dragon



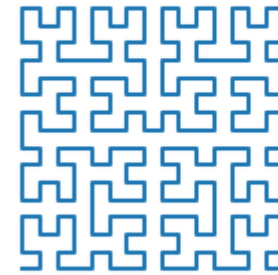
Axiom FX
 $X \rightarrow X+YF+$
 $Y \rightarrow -FX-Y$

Koch curve



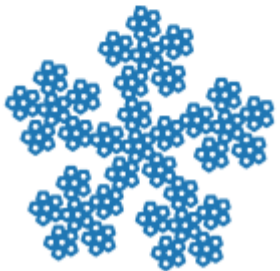
Axiom F
 $F \rightarrow F+F-F-F+F$

Hilbert curve



Axiom L
 $L \rightarrow +RF-LFL-FR+$
 $R \rightarrow -LF+RFR+FL-$

McWorter Pentigree curve



Axiom F-F-F-F-F
 $F \rightarrow F-F-F++F+F-F$

Turns 72°

Tree



Axiom F
 $F \rightarrow F[+FF][-FF]F[-F][+F]F$

Turns 36°
 [and] return to start point when done

Cesero fractal



Axiom F
 $F \rightarrow F+F--F+F$

Turns 80°

More space filling curves... (source code)

space-filling-L_systems.py

```
import matplotlib.pyplot as plt
from math import sin, cos, radians
```

```
def walk(commands,
         pos=(0, 0),
         forward=frozenset('F'),
         angle=0,
         turn=90):
    paths = [[pos]]
    stack = []
    for move in commands:
        if move in forward:
            pos = (pos[0]+cos(radians(angle)),
                  pos[1]+sin(radians(angle)))
            paths[-1].append(pos)
        elif move == '-': angle -= turn
        elif move == '+': angle += turn
        elif move == '[':
            stack.append((pos, angle))
        elif move == ']':
            pos, angle = stack.pop()
            paths.append([pos])
    return paths
```

```
def apply_rules(axiom, rules, repeat=1):
    for _ in range(repeat):
        axiom = ''.join(rules.get(symbol, symbol) for symbol in axiom)
    return axiom

curves = [ # Lindenmayer systems (L-systems)
    ('Sierpinski triangle', 'F-G-G', {'F': 'F-G+F+G-F', 'G': 'GG'}, 5, {'turn': 120, 'forward': {'F', 'G'}}),
    ('Sierpinski arrowhead curve', 'A', {'A': 'B-A-B', 'B': 'A+B+A'}, 5, {'turn': 60, 'forward': {'A', 'B'}}),
    ('Peano curve', 'L', {'L': 'LFRFL-F-RFLFR+F+LFRFL', 'R': 'RFLFR+F+LFRFL-F-RFLFR'}, 3, {}),
    ('Heighway dragon', 'FX', {'X': 'X+YF+', 'Y': '-FX-Y'}, 10, {}),
    ('Koch curve', 'F', {'F': 'F+F-F-F+F'}, 3, {}),
    ('Hilbert curve', 'L', {'L': '+RF-LFL-FR+', 'R': '-LF+RFR+FL-'}, 4, {}),
    ('McWorter Pentigree curve', 'F-F-F-F-F', {'F': 'F-F-F++F+F-F'}, 3, {'turn': 72}),
    ('Tree', 'F', {'F': 'F[+FF][-FF]F[-F][+F]F'}, 3, {'turn': 36}),
    ('Cesero fractal', 'F', {'F': 'F+F--F+F'}, 5, {'turn': 80})
]

for idx, (title, axiom, rules, repeat, walk_arg) in enumerate(curves, start=1):
    paths = walk(apply_rules(axiom, rules, repeat), **walk_arg)
    ax = plt.subplot(3, 3, idx, aspect='equal')
    ax.set_title(title)
    for path in paths:
        plt.plot(*zip(*path), '-')
plt.axis('off')
plt.show()
```

Relational data

- SQLite
- pandas



Two tables

Table: country			
name	population	area	capital
'Denmark'	5748769	42931	'Copenhagen'
'Germany'	82800000	357168	'Berlin'
'USA'	325719178	9833520	'Washington, D.C.'
'Iceland'	334252	102775	'Reykjavik'

Table: city			
name	country	population	established
'Copenhagen'	'Denmark'	775033	800
'Aarhus'	'Denmark'	273077	750
'Berlin'	'Germany'	3711930	1237
'Munich'	'Germany'	1464301	1158
'Reykjavik'	'Iceland'	126100	874
'Washington D.C.'	'USA'	693972	1790
'New Orleans'	'USA'	343829	1718
'San Francisco'	'USA'	884363	1776

SQL

pronounced ,ɛs,kjuːˈɛl or 'siːkwəl

- SQL = Structured Query **Language**
- **Database = collection of tables** stored persistently on disk
- ANSI and ISO standards since 1986 and 1987, respectively; origin early 70s
- Widespread used SQL databases (can handle many tables/rows/users):
[Oracle](#), [MySQL](#), [Microsoft SQL Server](#), [PostgreSQL](#) and [IBM DB2](#)
- **SQLite** is a very lightweight version storing a database in a **single file**, without a separate database server; first release in 2000
- SQLite is included in both iOS and Android mobil phones and operating systemes like Mac OS, Windows, and Fedora Linux

Table: country			
name	population	area	capital
'Denmark'	5748769	42931	'Copenhagen'
'Germany'	82800000	357168	'Berlin'
'USA'	325719178	9833520	'Washington, D.C.'
'Iceland'	334252	102775	'Reykjavik'



The Course "[Databases](#)" gives a more in-depth introduction to SQL (MySQL)

Examples : SQL statements

Table: country			
name	population	area	capital
'Denmark'	5748769	42931	'Copenhagen'
'Germany'	82800000	357168	'Berlin'
'USA'	325719178	9833520	'Washington, D.C.'
'Iceland'	334252	102775	'Reykjavik'

- CREATE TABLE country (name, population, area, capital)
- INSERT INTO country VALUES ('Denmark', 5748769, 42931, 'Copenhagen')
- UPDATE country SET population=5748770 WHERE name='Denmark'
- SELECT name, capital FROM country WHERE population >= 1000000
> [('Denmark', 'Copenhagen'), ('Germany', 'Berlin'), ('USA', 'Washington, D.C.')]
- SELECT * FROM country WHERE capital = 'Berlin'
> [('Germany', 82800000, 357168, 'Berlin')]
- SELECT country.name, city.name, city.established FROM city, country
WHERE city.name=country.capital AND city.population < 700000
> [('Iceland', 'Reykjavik', 874), ('USA', 'Washington, D.C.', 1790)]
- DELETE FROM country WHERE name = 'Germany'
- DROP TABLE country

sqlite-example.py

```
import sqlite3

connection = sqlite3.connect('example.sqlite') # creates file if necessary
c = connection.cursor()

c.executescript('''DROP TABLE IF EXISTS country; -- multiple SQL statements
                  DROP TABLE IF EXISTS city''')

countries = [('Denmark', 5748769, 42931, 'Copenhagen'),
             ('Germany', 82800000, 357168, 'Berlin'),
             ('USA', 325719178, 9833520, 'Washington, D.C.'),
             ('Iceland', 334252, 102775, 'Reykjavik')]

cities = [('Copenhagen', 'Denmark', 775033, 800),
          ('Aarhus', 'Denmark', 273077, 750),
          ('Berlin', 'Germany', 3711930, 1237),
          ('Munich', 'Germany', 1464301, 1158),
          ('Reykjavik', 'Iceland', 126100, 874),
          ('Washington, D.C.', 'USA', 693972, 1790),
          ('New Orleans', 'USA', 343829, 1718),
          ('San Francisco', 'USA', 884363, 1776)]

c.execute('CREATE TABLE country (name, population, area, capital)')
c.execute('CREATE TABLE city (name, country, population, established)')
c.executemany('INSERT INTO country VALUES (?, ?, ?, ?)', countries)
c.executemany('INSERT INTO city VALUES (?, ?, ?, ?)', cities)

connection.commit() # save data to database before closing
connection.close()
```

SQLite

SQLite query examples

try to avoid using the asterisk (*) as a good habit

www.sqlitetutorial.net/sqlite-select

sqlite-example.py

```
for row in c.execute('SELECT * FROM country'): # * = all columns, execute returns iterator
    print(row)                                # row is by default a Python tuple

for row in c.execute('''SELECT * FROM city, country -- all pairs of rows from city x country
                        WHERE city.name = country.capital AND city.population < 700000'''):
    print(row)

print(*c.execute('''SELECT country.name,
                    COUNT(city.name) AS cities,
                    100 * SUM(city.population) / country.population
                    FROM city JOIN country ON city.country = country.name -- SQL join 2 tables
                    WHERE city.population > 500000 -- only consider big cities
                    GROUP BY city.country -- output has one row per group of rows
                    ORDER BY cities DESC, SUM(city.population) DESC''')) # ordering of output
```

Python shell

```
| ('Denmark', 5748769, 42931, 'Copenhagen')
| ('Germany', 82800000, 357168, 'Berlin')
| ('USA', 325719178, 9833520, 'Washington, D.C.')
| ('Iceland', 334252, 102775, 'Reykjavik')

| ('Reykjavik', 'Iceland', 126100, 874, 'Iceland', 334252, 102775, 'Reykjavik')
| ('Washington, D.C.', 'USA', 693972, 1790, 'USA', 325719178, 9833520, 'Washington, D.C.')
| ('Germany', 2, 6) ('USA', 2, 0) ('Denmark', 1, 13)
```

SQL injection

Right way

```
c.execute('INSERT INTO users VALUES (?)', (user,))
```

unsafe-example.py

```
import sqlite3
connection = sqlite3.connect('users.sqlite')
c = connection.cursor()
c.execute('CREATE TABLE users (name)')
while True:
    user = input('New user: ')
    c.executescript(f'INSERT INTO users VALUES ("{user}")')
    connection.commit()
    print(list(c.execute('SELECT * FROM users')))
```

can execute a string
containing several
SQL statements



Insecure
NEVER use f-string
with user input

Python shell

```
> New user: gerth
| [('gerth',)]
> New user: guido
| [('gerth',), ('guido',)]
> New user: evil"); DROP TABLE users; --
| sqlite3.OperationalError: no such table: users
```

INSERT INTO users VALUES ("evil"); DROP TABLE users; --")

HI, THIS IS
YOUR SON'S SCHOOL.
WE'RE HAVING SOME
COMPUTER TROUBLE.



OH, DEAR - DID HE
BREAK SOMETHING?
IN A WAY -



DID YOU REALLY
NAME YOUR SON
Robert'); DROP
TABLE Students;-- ?



OH. YES. LITTLE
BOBBY TABLES,
WE CALL HIM.

WELL, WE'VE LOST THIS
YEAR'S STUDENT RECORDS.
I HOPE YOU'RE HAPPY.



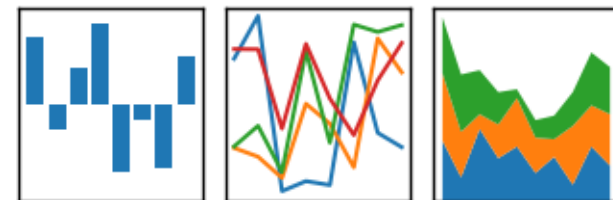
AND I HOPE
YOU'VE LEARNED
TO SANITIZE YOUR
DATABASE INPUTS.

Pandas



- Comprehensive Python library for data manipulation and analysis, in particular tables and time series
- Pandas **data frames** = tables
- Supports interaction with SQL, CSV, JSON, ...
- Integrates with Jupyter, numpy, matplotlib, ...

pandas
 $y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$



Pandas integration with Jupyter

- Tables (Pandas data frames) are rendered nicely in Jupyter

```
In [1]: 1 import pandas as pd
        2 students = pd.read_csv('students.csv')
        3 students
```

Out [1]:

	Name	City
0	Donald Duck	Copenhagen
1	Goofy	Aarhus
2	Mickey Mouse	Aarhus

students.csv

```
Name, City
"Donald Duck", "Copenhagen"
"Goofy", "Aarhus"
"Mickey Mouse", "Aarhus"
```

Reading tables (data frames)

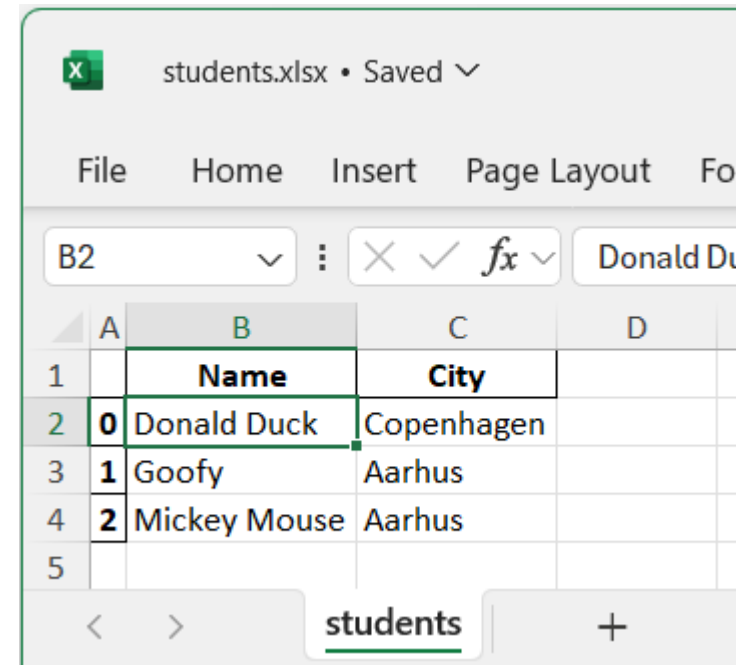
- Pandas provides functions for reading and writing pandas.DataFrames as files in different data formats, e.g. SQLite, csv and EXCEL files

pandas-example.py

```
import pandas as pd
import sqlite3
connection = sqlite3.connect('example.sqlite')
countries = pd.read_sql_query('SELECT * FROM country', connection)
cities = pd.read_sql_query('SELECT * FROM city', connection)
students = pd.read_csv('students.csv')
students.to_sql('students', connection, if_exists='replace')
students.to_excel('students.xlsx', sheet_name='students')
print(students)
```

Python shell

	Name	City
0	Donald Duck	Copenhagen
1	Goofy	Aarhus
2	Mickey Mouse	Aarhus



students.xlsx • Saved

File Home Insert Page Layout Formulas

B2 X ✓ fx Donald Duck

	A	B	C	D
1		Name	City	
2	0	Donald Duck	Copenhagen	
3	1	Goofy	Aarhus	
4	2	Mickey Mouse	Aarhus	
5				

< > students +

Selecting columns and rows

Table: country			
name	population	area	capital
'Denmark'	5748769	42931	'Copenhagen'
'Germany'	82800000	357168	'Berlin'
'USA'	325719178	9833520	'Washington, D.C.'
'Iceland'	334252	102775	'Reykjavik'

Python shell

```
> countries['name']           # select column
> countries.name              # same as above
> countries[['name', 'capital']] # select multiple columns, note double-[]
> countries.head(2)           # first 2 rows
> countries[1:3]               # slicing rows, rows 1 and 2
> countries[::2]               # slicing rows, rows 0 and 2
> countries.at[1, 'area']      # indexing cell by (row label, column name)
> cities[(cities['name'] == 'Berlin') | (cities['name'] == 'Munich')] # select rows
|      name  country  population  established
|  2  Berlin  Germany    3711930         1237 # note original row labels
|  3  Munich  Germany    1464301         1158
> pd.DataFrame([[1, 2], [3, 4], [5, 6]], columns=['x', 'y']) # create DF from list
> pd.DataFrame(np.random.random((3, 2)), columns=['x', 'y']) # from numpy
```

Row labels

Python shell

```
> df = pd.DataFrame(np.arange(1, 13).reshape(3, 4),
                    index=['q', 'w', 'e'],          # row labels
                    columns=['c', 'a', 'd', 'e'])    # column names

> df
|   c   a   d   e
| q   1   2   3   4 # row labels can be strings
| w   5   6   7   8
| e   9  10  11  12

> df.loc['w':'e', ['e', 'a']] # slice of labeled rows
|   e   a
| w   8   6
| e  12  10

> df.loc['w'] # single row (a one-dimensional pd.Series)
| c     5
| a     6
| d     7
| e     8
| Name: w, dtype: int32

> df.iloc[:2, :2] # use iloc to work with integer indexes
|   c   a
| q   1   2
| w   5   6
```


Merging tables and creating a new column

pandas-example.py

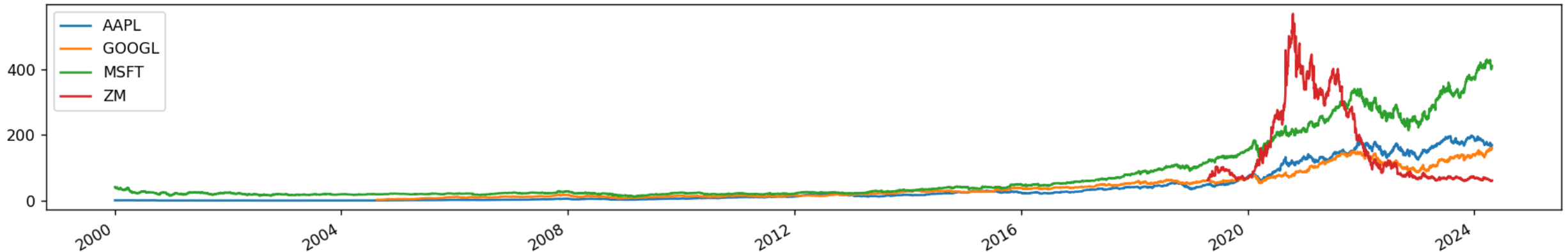
```
M = pd.merge(countries, cities, left_on='capital', right_on='name')
# both data frames had a 'name' and 'population' column
M1 = M.rename(columns={
    'population_x': 'country_population',
    'population_y': 'capital_population'
})
M2 = M1.drop(columns=['name_x', 'name_y'])
M2['%pop in capital'] = M2.capital_population / M2.country_population # add column
M2.sort_values('%pop in capital', ascending=False, inplace=True)
print(M2[['country', '%pop in capital']])
```

Python shell

	country	%pop in capital	
3	Iceland	0.377260	# note row labels are permuted
0	Denmark	0.134817	
1	Germany	0.044830	
2	USA	0.002131	

Pandas datareader and Matplotlib

- pandas_datareader provides access to many data sources
- dataframes have a .plot method (using matplotlib.pyplot)
- `pip install pandas-datareader` and `pip install setuptools`



pandas-datareader.py

```
import matplotlib.pyplot as plt
import pandas_datareader
#df = pandas_datareader.data.DataReader(['AAPL', 'GOOGL', 'MSFT', 'ZM'], 'stooq') # ignores start=...
df = pandas_datareader.stooq.StooqDailyReader(['AAPL', 'GOOGL', 'MSFT', 'ZM'], start='2000-01-01').read()
df['Close'].plot()
plt.legend()
plt.show()
```

pandas-datareader.readthedocs.io

pandas-datareader.readthedocs.io/en/latest/readers/stooq.html

Hierarchical / Multi-level indexing (MultiIndex)

Python shell

```
> df.tail(2)
| Attributes      Close      ...      Volume
| Symbols        AAPL      GOOGL      MSFT      ...      GOOGL      MSFT      ZM
| Date      ...
| 2020-04-29      287.73      1342.18      177.43      ...      5417888.0      51286559.0      22033320.0
| 2020-04-30      293.80      1346.70      179.21      ...      2788644.0      53627543.0      16648922.0
> df['Close'].tail(2)
| Symbols      AAPL      GOOGL      MSFT      ZM
| Date
| 2020-04-29      287.73      1342.18      177.43      146.48
| 2020-04-30      293.80      1346.70      179.21      135.17
> df['Close']['GOOGL'].tail(2)
| Date
| 2020-04-29      1342.18
| 2020-04-30      1346.70
| Name: GOOGL, dtype: float64
> df.loc[:, pd.IndexSlice[:, 'GOOGL']].tail(2)
| Attributes      Close      High      Low      Open      Volume
| Symbols        GOOGL      GOOGL      GOOGL      GOOGL      GOOGL
| Date
| 2020-04-29      1342.18      1360.15      1326.73      1345.00      5417888.0
| 2020-04-30      1346.70      1350.00      1321.50      1331.36      2788644.0
```

Both rows and columns can have multi-level indexing

Python shell

```
> df.columns
| MultiIndex([( 'Close', 'AAPL'),
|             ( 'Close', 'GOOGL'),
|             ( 'Close', 'MSFT'),
|             ( 'Close', 'ZM'),
|             ( 'High', 'AAPL'),
|             ( 'High', 'GOOGL'),
|             ( 'High', 'MSFT'),
|             ( 'High', 'ZM'),
|             ( 'Low', 'AAPL'),
|             ( 'Low', 'GOOGL'),
|             ( 'Low', 'MSFT'),
|             ( 'Low', 'ZM'),
|             ( 'Open', 'AAPL'),
|             ( 'Open', 'GOOGL'),
|             ( 'Open', 'MSFT'),
|             ( 'Open', 'ZM'),
|             ( 'Volume', 'AAPL'),
|             ( 'Volume', 'GOOGL'),
|             ( 'Volume', 'MSFT'),
|             ( 'Volume', 'ZM')],
|            names=['Attributes', 'Symbols'])
```

THIS IS YOUR MACHINE LEARNING SYSTEM?

YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.

WHAT IF THE ANSWERS ARE WRONG?

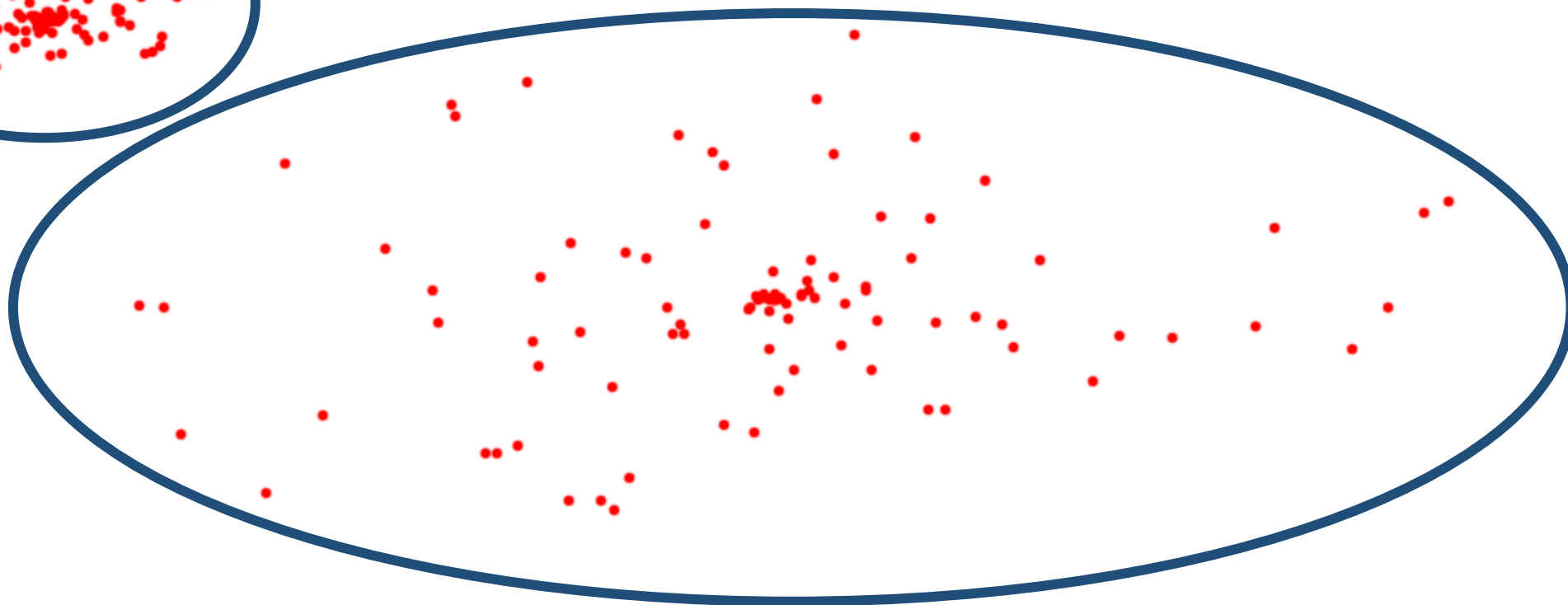
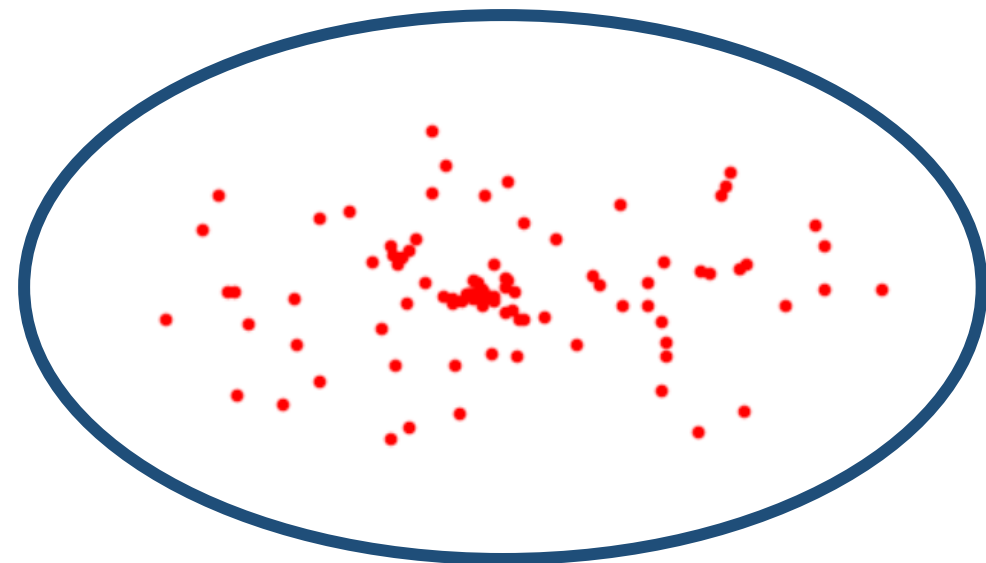
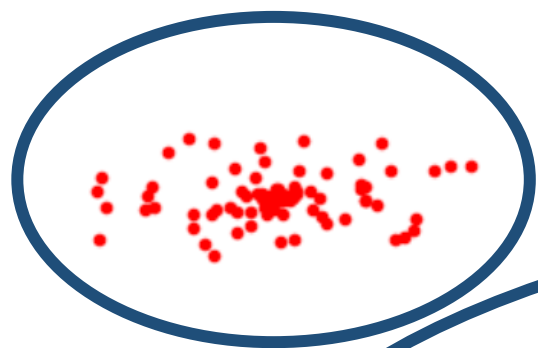
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.



Clustering

- k-means
- `scipy.cluster.vq.kmeans`
- DBSCAN*
- neural networks

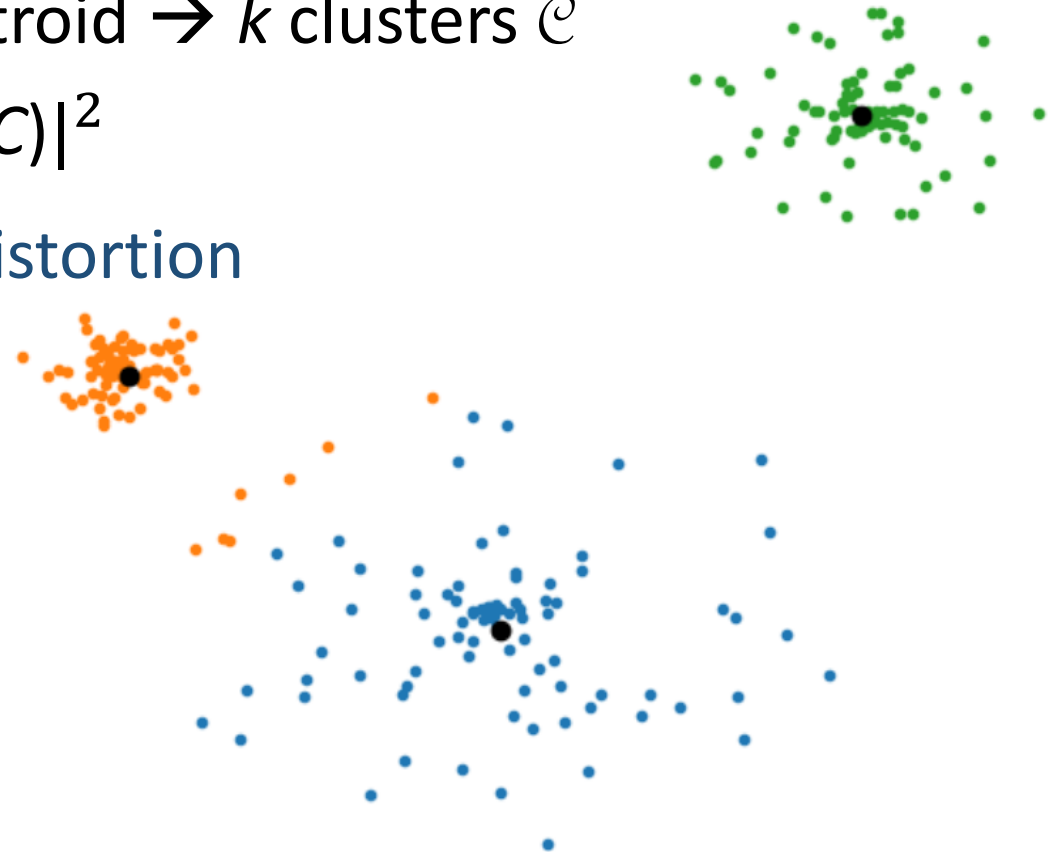
3 clusters / groups of points



Clustering = Optimization problem

Example: *k*-means

- Given n input points and an integer k , find k *centroid* points
- Assign each input point to nearest centroid $\rightarrow k$ clusters $\mathcal{C}$
- *distortion* = $\sum_{C \in \mathcal{C}} \sum_{p \in C} |p - \text{centroid}(C)|^2$
- **Goal** : Find k centroids that minimize distortion

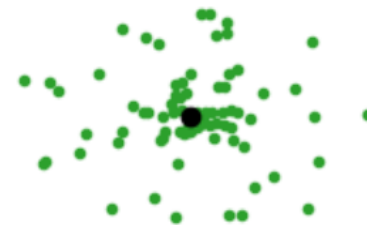


k-means for $k = 1$

- Let the centroid point c for a point set C be the point minimizing the

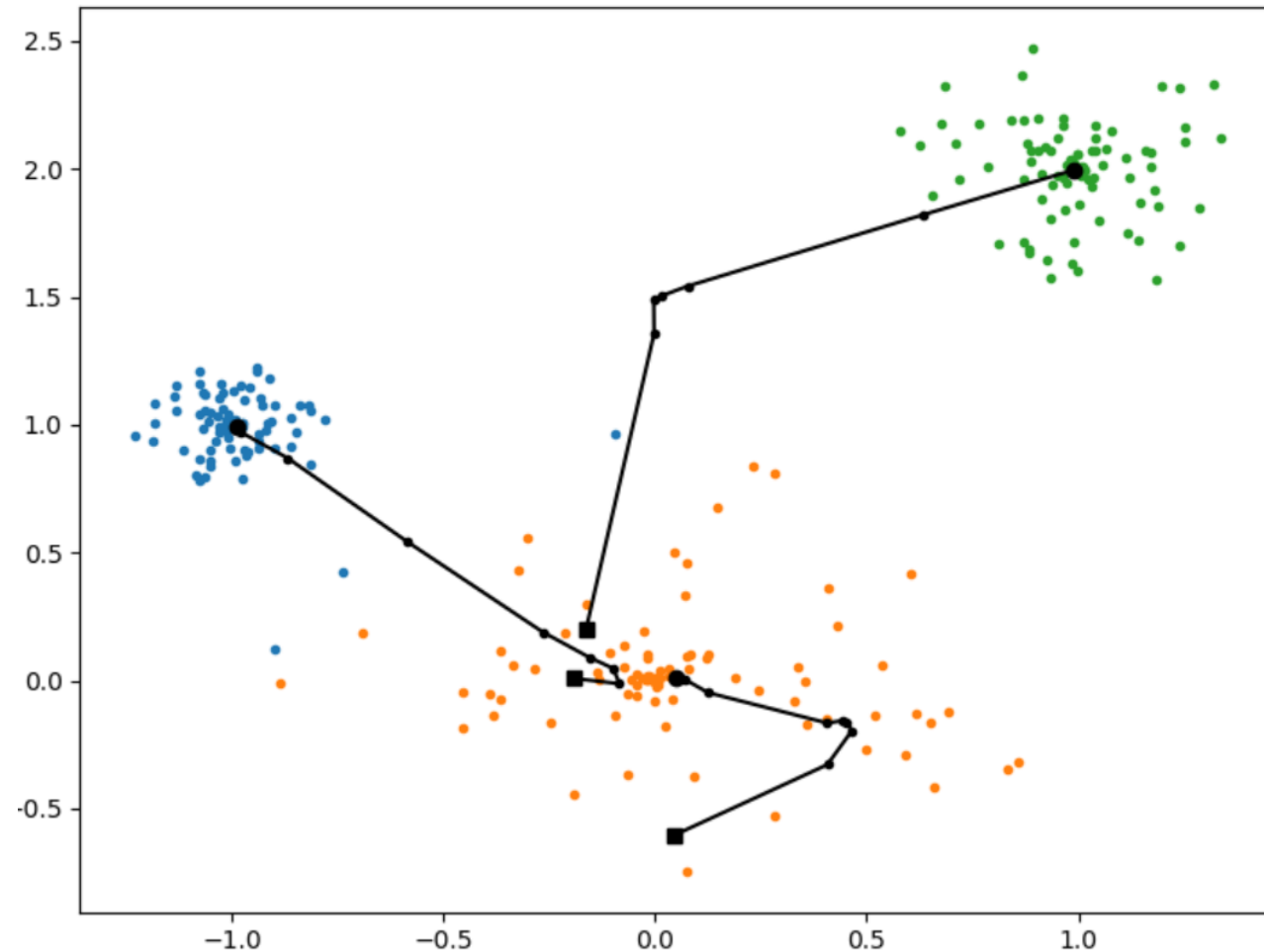
$$\text{distortion} = \sum_{p \in C} |p - c|^2$$

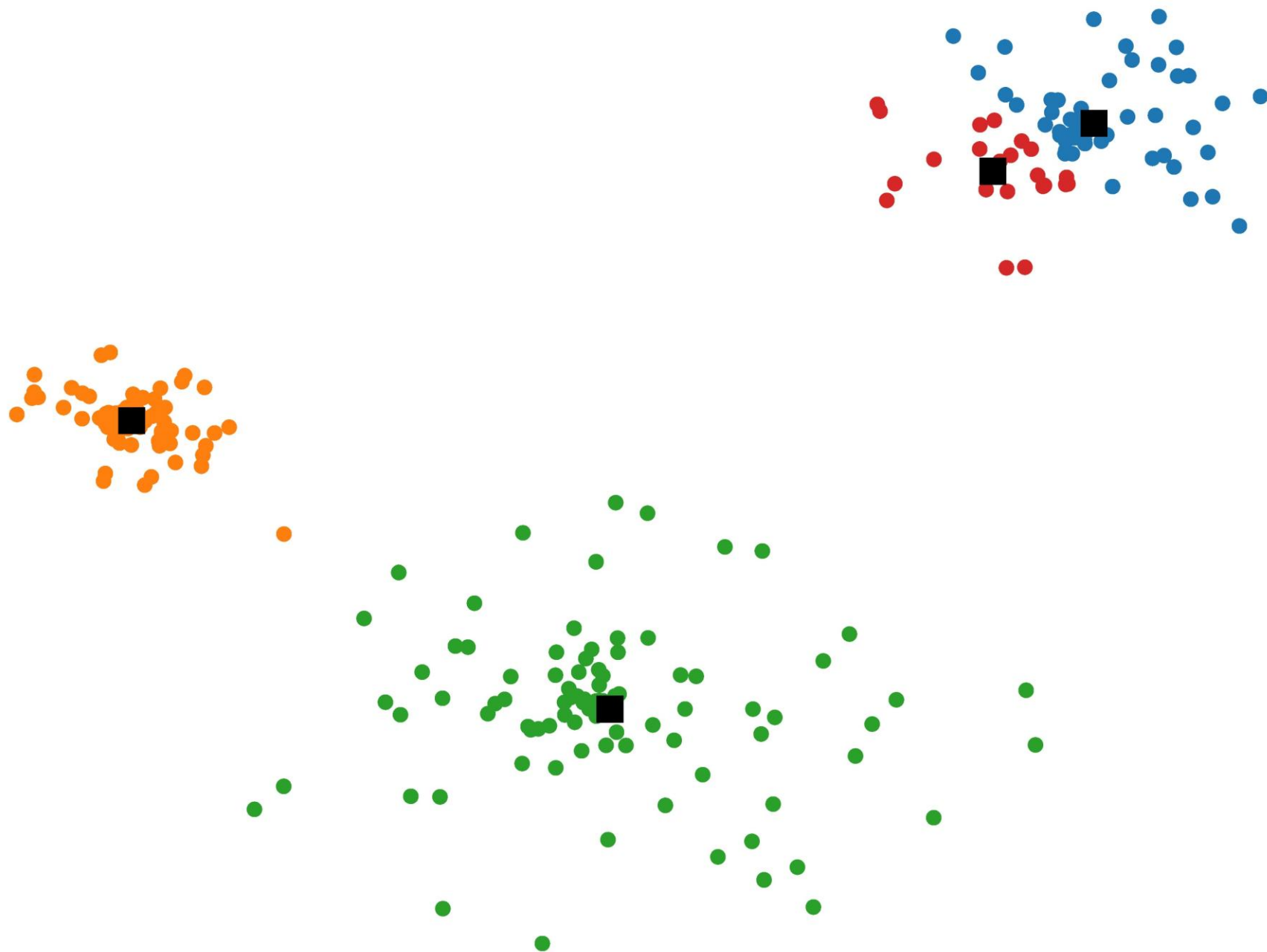
- Theorem $c = \text{average}(C)$

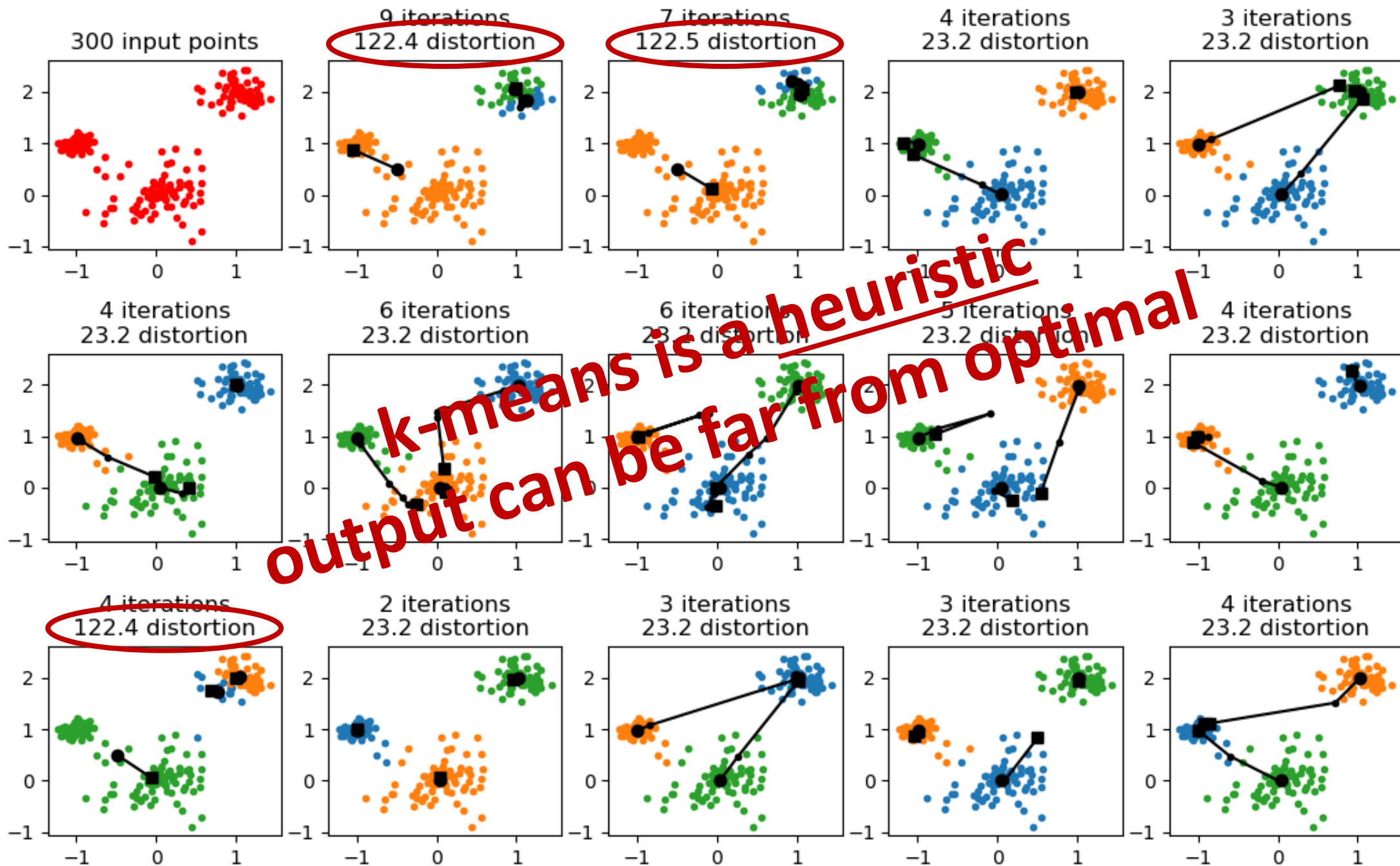


k-means - Lloyd's method (pseudo code)

```
centroids = k distinct random input points  
while centroids change:  
    create clusters C by assigning points to the nearest centroid  
    centroids = average of each cluster
```







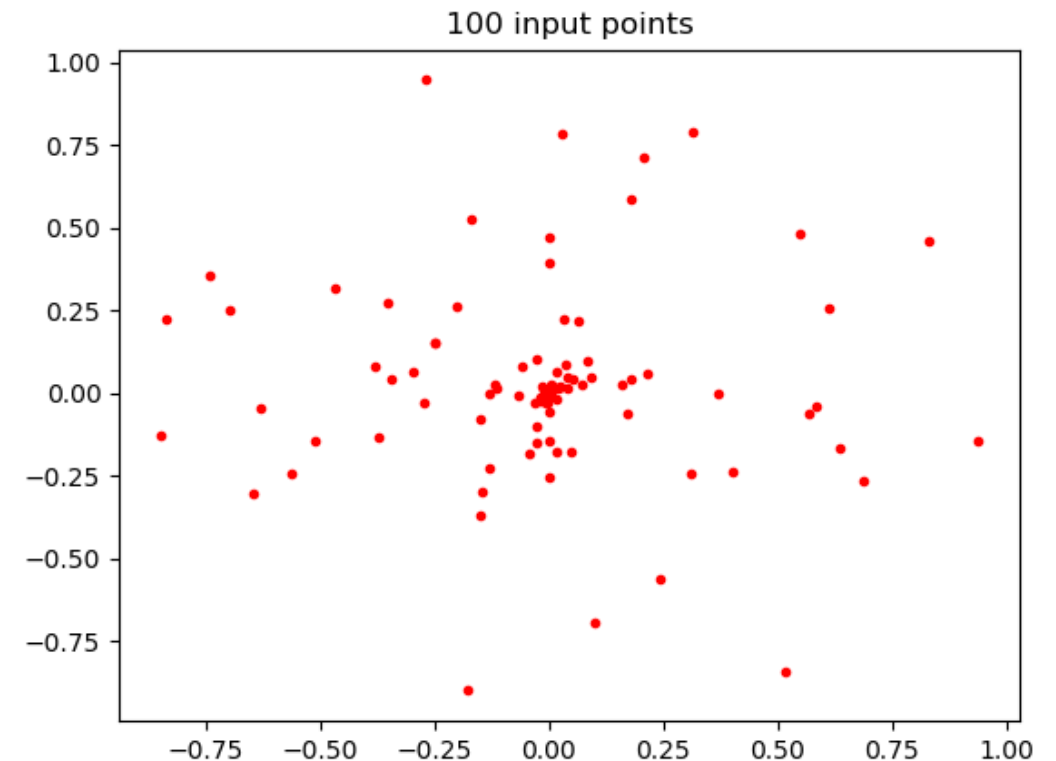
Generating random points (just one random approach)

`k_means.py`

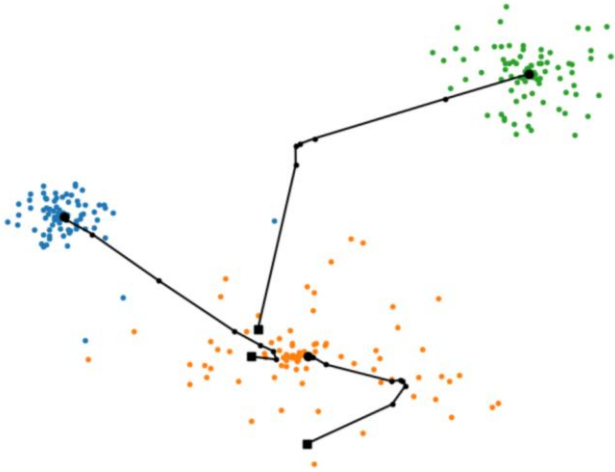
```
from random import random
from math import pi, cos, sin

def random_point(x, y, radius):
    angle = 2 * pi * random()
    r = radius * random() ** 2
    return x + r * cos(angle), y + r * sin(angle)

def random_points(n, x, y, radius):
    for _ in range(n):
        yield random_point(x, y, radius)
```



k-means



`k_means.py`

```
from random import sample
from numpy import argmin, mean

def dist(p, q):
    return sum((a - b)**2 for a, b in zip(p, q))

def k_means(points, k):
    centroid = sample(points, k)
    centroids = [ centroid ] # history for visualization
    while True:
        clusters = [[] for _ in centroid]
        for p in points:
            i = argmin([dist(p, c) for c in centroid])
            clusters[i].append(p)

        centroid = [tuple(map(mean, zip(*c))) for c in clusters]
        if centroid == centroids[-1]:
            break

        centroids.append(centroid)
        if any(len(c) == 0 for c in clusters):
            print('Not good - empty cluster')
            break

    return clusters
```

k-mean limitations

- Can easily converge to a solution far from a global minimum 
 - Solution – try several times and take the best (possibly since we can measure the quality (= distortion) of a solution)
- Clusters can become empty
 - Solution – discard and restart / take a random point out as a new centroid / take point furthest away from existing centroids /
- Sensitive to the scales of the different dimensions
 - Solution – apply some kind of initial normalization of coordinates

k-means - better bounds

- The **k-means++** algorithm achieves an **expected guarantee** to be at most a factor $8(2 + \ln k)$ from the optimal [Vassilvitskii & Arthur]
- There exist **polynomial time approximation schemes** that find a solution that is guaranteed $1 + \epsilon$ of the optimal (but running time exponential in k and dimension of points) [Har-Peled et al.]
- **In practice: A heuristic is most often the algorithm of choice**

scipy.cluster.vq.kmeans

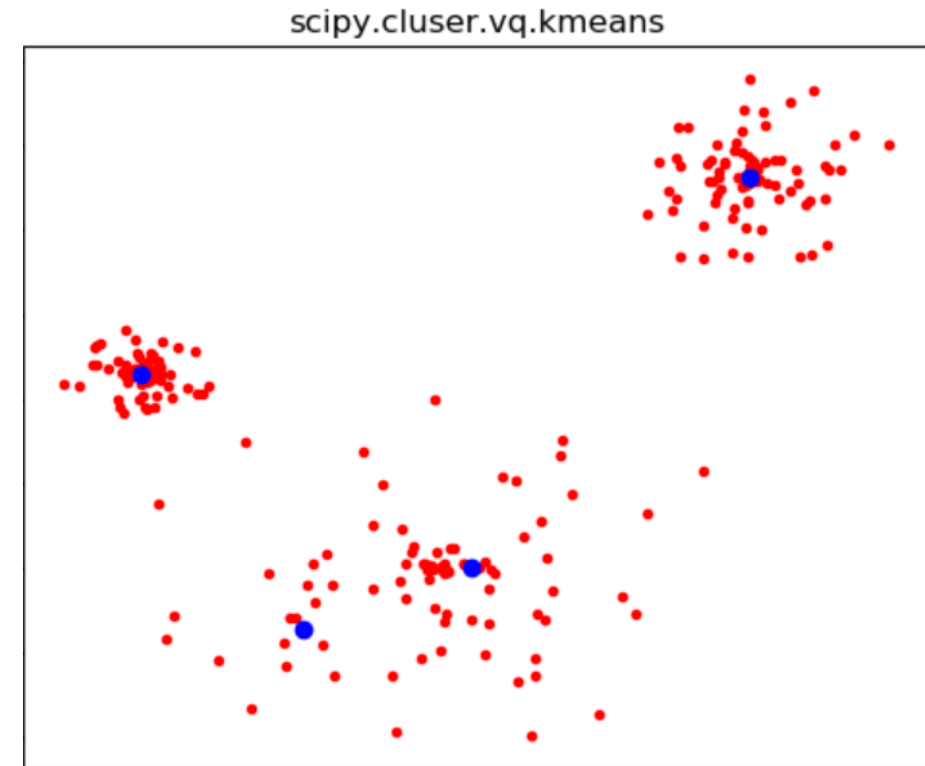
k_means.py

```
from scipy.cluster.vq import kmeans, whiten
import matplotlib.pyplot as plt

points = whiten(points) # normalize variance of points
centroids, distortion = kmeans(points, K)

plt.plot(*zip(*points), 'r.')
plt.plot(*zip(*centroids), 'bo')
plt.title('scipy.cluster.vq.kmeans')
plt.show()
```

Note: According to the documentation "whiten must be called prior to passing an observation matrix to kmeans"



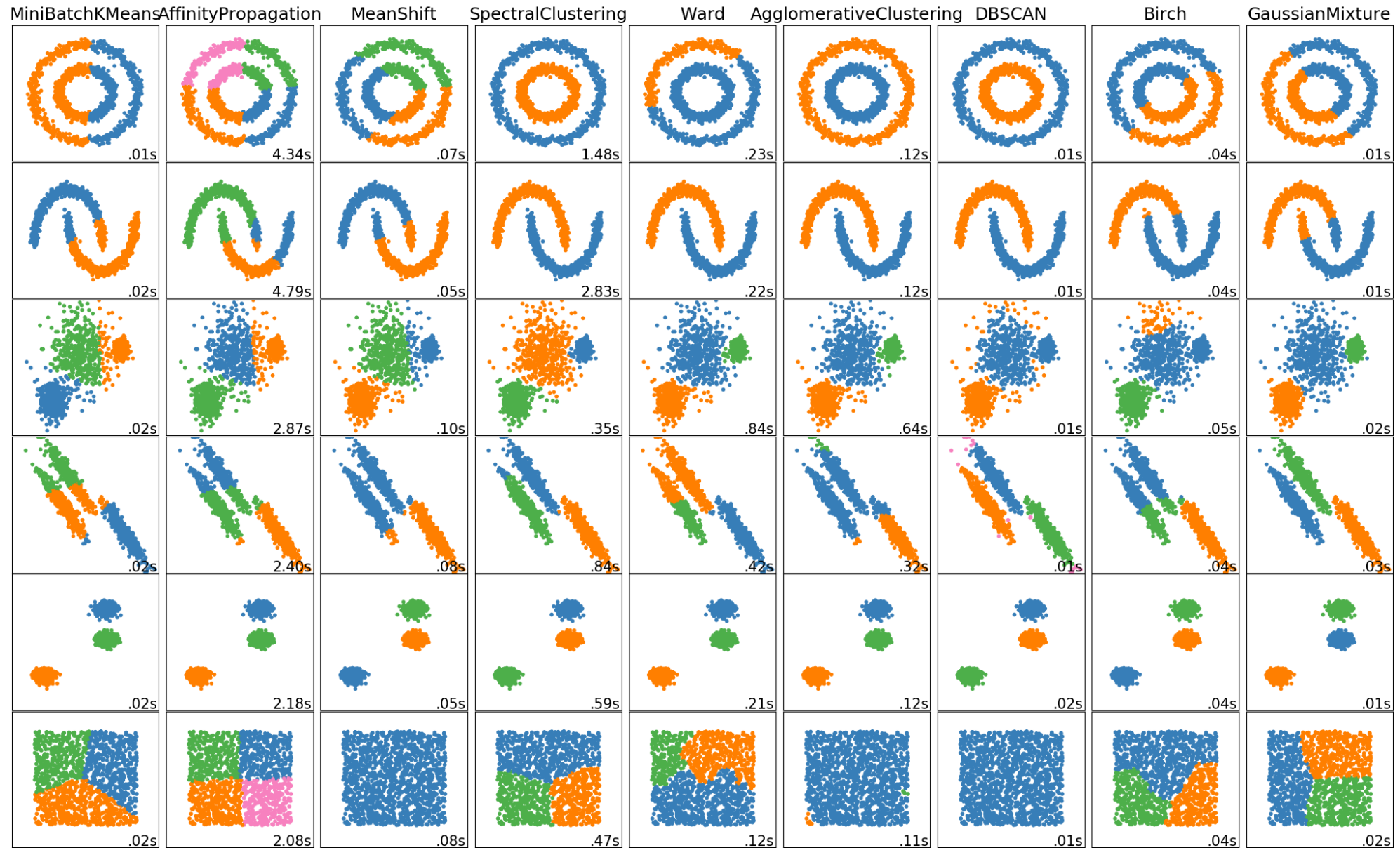
scipy.cluster.vq.whiten

- Normalizes / scales each dimension to have unit variance 1.0

$$\text{Var}(X) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$$

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i$$

Other Python clustering methods - `sklearn.cluster`



DBSCAN*

dbscan.py

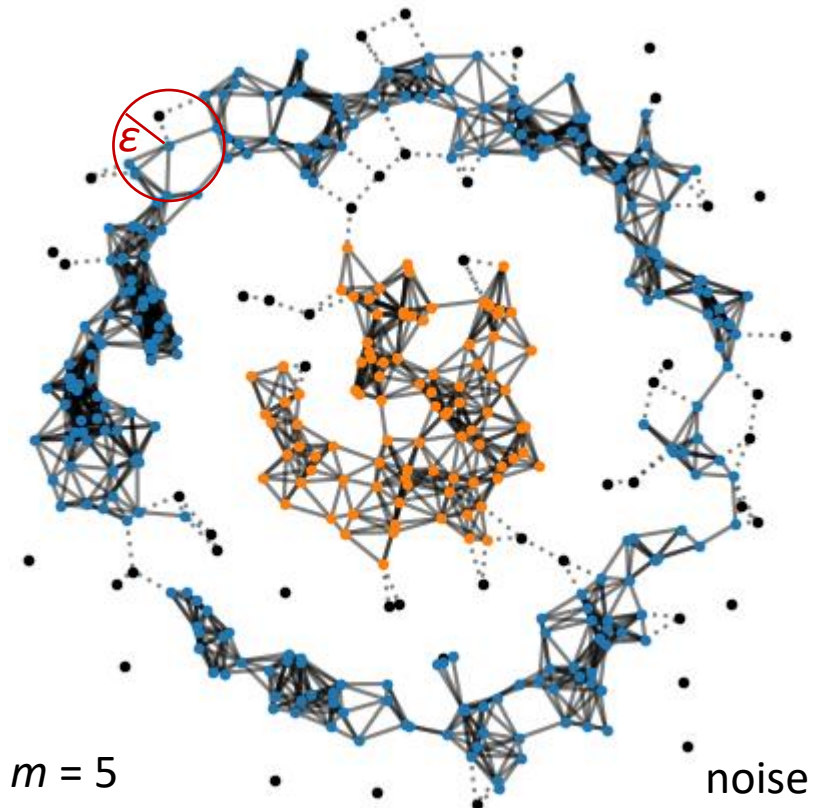
```
def dbscan(points, epsilon, m):
    def dist(p, q):
        return sum((pi - qi) ** 2 for pi, qi in zip(p, q))

    def close(p, q):
        return dist(p, q) <= epsilon ** 2

    core, noise, clusters = [], [], []
    for p in points:
        if sum(close(p, q) for q in points) >= m:
            core.append(p)
        else:
            noise.append(p)

    while core:
        cluster = [core.pop()]
        for p in cluster:
            for q in list(core):
                if close(p, q):
                    cluster.append(q)
                    core.remove(q)
        clusters.append(cluster)
    return clusters, noise
```

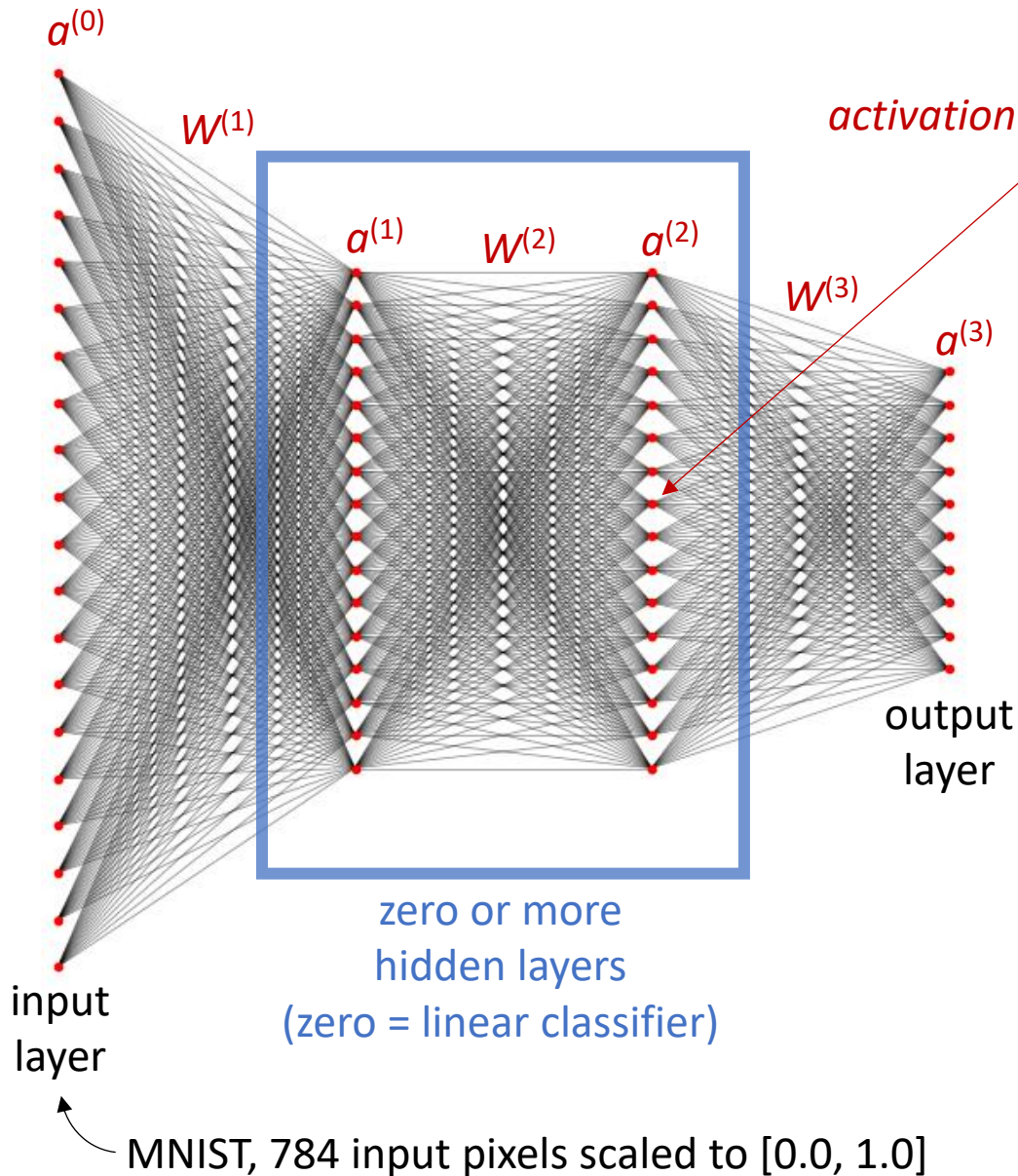
- Parameters ϵ and m
- p is a **core point** when
$$|\{q \mid |p - q| \leq \epsilon\}| \geq m$$
- Remaining points are **noise**
- Core points p and q are in the same **cluster** if $|p - q| \leq \epsilon$



Data Mining Algorithms

- k-means, DBSCAN*, and more generally clustering, is just one field in the area of *Data Mining*
- For more information see the webpage Top 10 Data Mining Algorithms, Explained a follow up to the below paper
- X. Wu et al., *Top 10 algorithms in data mining*, Knowledge and Information Systems, 14(1):1–37, 2008. DOI 10.1007/s10115-007-0114-2

Neural networks (one slide introduction)



activation

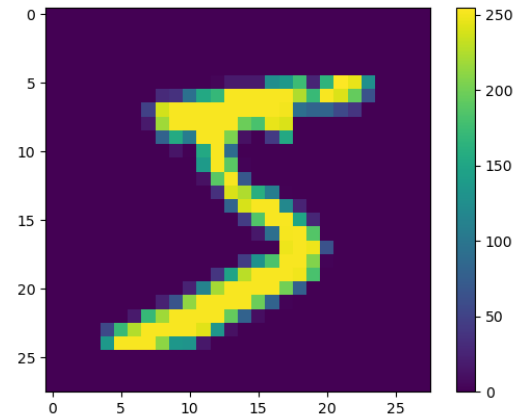
$$a_i^{(l)} = f^{(l)} \left(\sum_j a_j^{(l-1)} \cdot W_{ji}^{(l)} + b_i^{(l)} \right)$$

activation function (nonlinear)

weight

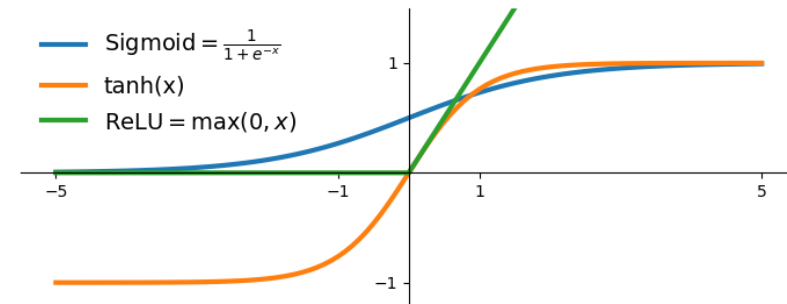
bias

Classification, like MNIST,
prediction = index of node
with maximum output



MNIST : 28 x 28 pixel
values from [0, 255]

Common activation functions



e.g. mean squared error

$$\frac{1}{n} \sum_{(x,y)} |\text{out}(x) - y|^2$$

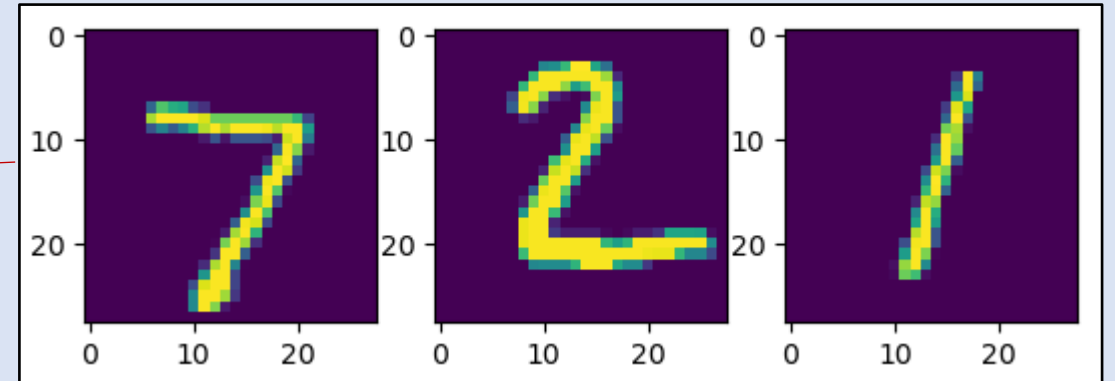
Learning

Find W s and b s performing well (minimize a cost function) on a set of n training inputs x with known output y using *backpropagation* / *stochastic gradient descent*

Applying a linear classifier using Numpy: $x \cdot W + b$

Python shell

```
| import matplotlib.pyplot as plt
| import numpy as np
| from tensorflow import keras
| (train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.load_data()
| type(test_images)
> <class 'numpy.ndarray'>
| test_images.shape
> (10000, 28, 28) # 10_000 images 28 x 28
| test_labels.shape
> (10000,) # 10_000 labels
| test_labels[:3]
> array([7, 2, 1], dtype=uint8) manually generated labels
| for i, image in zip(range(3), test_images):
|     plt.subplot(1, 3, i + 1)
|     plt.imshow(image)
| plt.show()
| W, b = map(np.array, eval(open('mnist_linear.weights').read())) # read W and b from file
| print(W.shape, W.dtype, b.shape, b.dtype)
> (784, 10) float64 (10,) float64
| print([np.argmax(image.reshape(28 * 28) @ W + b) for image in test_images[:3]])
> [7, 2, 1] # correct on 9_142 of the 10_000 images for the above file, ie accuracy 91%
```



Graphical user interfaces (GUI)

- Tkinter

primitive_calculator.py

```
accumulator = 0

while True:
    print('Accumulator:', accumulator)
    print('Select:')
    print('  1: clear')
    print('  2: add')
    print('  3: subtract')
    print('  4: multiply')
    print('  5: quit')

    choice = int(input('Choice: '))

    match choice:
        case 1: accumulator = 0
        case 2: accumulator += int(input('add: '))
        case 3: accumulator -= int(input('subtract: '))
        case 4: accumulator *= int(input('multiply by: '))
        case 5: break
```

Python shell

```
Accumulator: 0
Select:
  1: clear
  2: add
  3: subtract
  4: multiply
  5: quit
Choice: 2
add: 10
Accumulator: 10
Select:
  1: clear
  2: add
  3: subtract
  4: multiply
  5: quit
Choice: 2
add: 15
Accumulator: 25
Select:
...
```

Homepage - Introduktion til prcX

← → ↺ 🔒 🌐 https://brightspace.au.dk/d2l/home/539



Introduktion til programmering me

Course Home Content Course Tools ▾ Classlist

```
def memoize(f):  
    # answers[args] = f(*args)  
    answers = {}  
    def wrapper(*args):  
        # if args not in answers:  
        #     answers[args] = f(*args)  
        return answers[args]  
    return wrapper
```



This course is open for studie

AutoSave On gui.p... - Sa... Gerth Støtting Brodal GS

File Home AU Insert Design Transitions Animations Slide Show Review View Help Acrobat

Templafy Paste Slides Font Paragraph Drawing Editing Create and Share Adobe PDF Dictate Design Ideas

Templafy Clipboard Adobe Acrobat Voice Designer

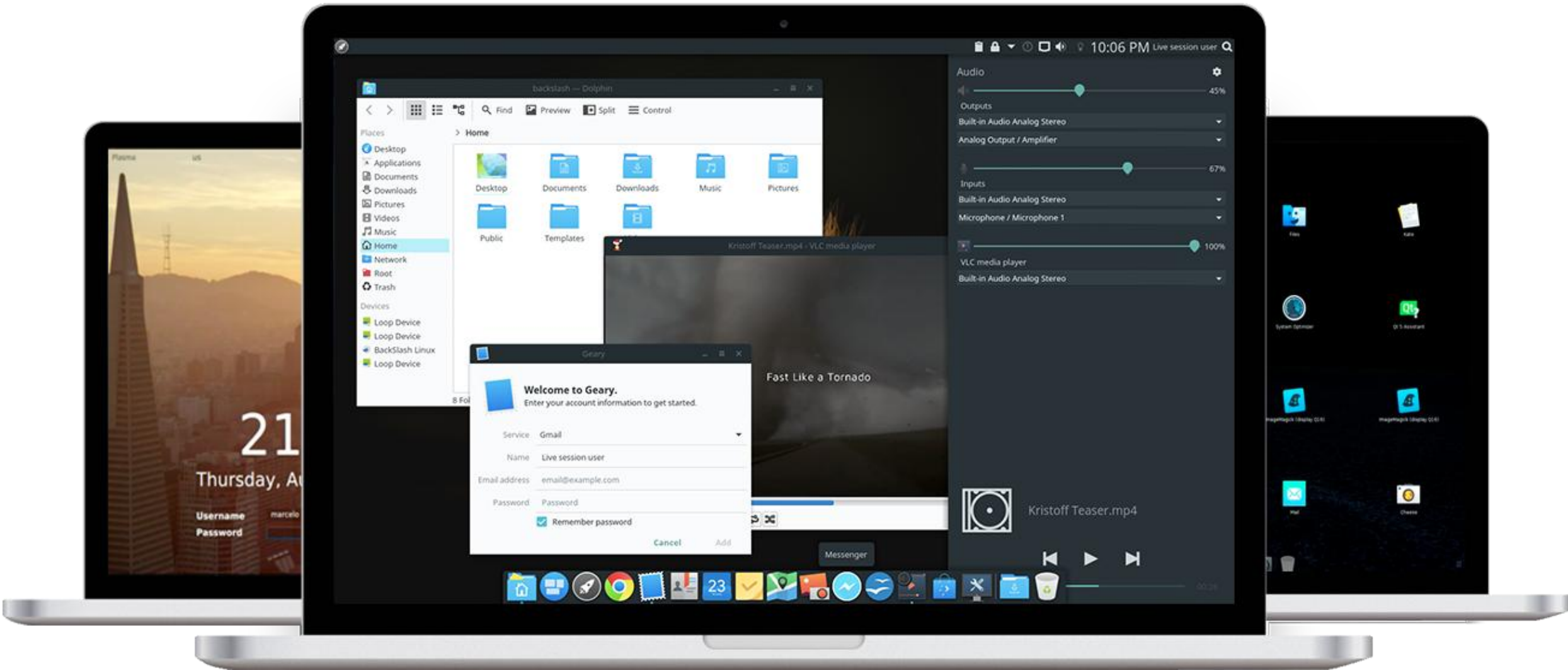
1 2 3 4 5 6 7 8 9

Graphical user interfaces (GUI)

- Tkinter

Click to add notes

Slide 1 of 19 Danish Notes Display Settings 48 %



BackSlash Linux GUI
www.backslashlinux.com

Python GUI's (Graphical Users Interfaces)

- There is a long list of GUI frameworks and toolkits, designer tools
 - we will only briefly look at **Tkinter**
- GUI are, opposed to a text terminal, **easier to use, more intuitive** and **flexible**
- Windows, icons, menus, buttons, scrollbars mouse / touch / keyboard interaction etc.
- Operating system (e.g. Windows, macOS, iOS, Linux, Android) provides basic functionality in particular a **window manager**
- Writing GUI applications from scratch can be painful – frameworks try to provide all standard functionality



en.wikipedia.org/wiki/Colossal_Cave_Adventure

Tkinter

- “Tkinter is Python's de-facto standard GUI (Graphical User Interface) package. It is a thin object-oriented layer on top of Tcl/Tk.”
- “Tcl is a high-level, general-purpose, interpreted, dynamic programming language.”
- “Tk is a free and open-source, cross-platform widget toolkit that provides a library of basic elements of GUI widgets for building a graphical user interface (GUI) in many programming languages.”
- “The popular combination of Tcl with the Tk extension is referred to as Tcl/Tk, and enables building a graphical user interface (GUI) natively in Tcl. Tcl/Tk is included in the standard Python installation in the form of Tkinter.”

Terminology

- **widgets** (e.g. buttons, editable text fields, labels, scrollbars, menus, radio buttons, check buttons, canvas for drawing, frames...)
- **events** (e.g. key press, mouse click, mouse entering/leaving, resizing windows, redraw requests, ...)
- **listening** (application waits for events to be fired)
- **event handler** (a function whose purpose is to handle an event, many triggered by user or OS/Window manager)
- **geometry managers** (how to organize widgets in a window: Tkinter *pack, grid, place*)

docs.python.org/3/library/tk.html



“tkinter is also famous for having an outdated look and feel”

- Comes with Python
- Alternatives PySide, PyQt, Kivy, ...

Welcome example



welcome.py

```
import tkinter
root = tkinter.Tk() # root window
def do_quit(): # event handler for 'Close' button
    root.destroy()
root.title('Tkinter Welcome GUI')
label = tkinter.Label(root, text='Welcome to Tkinter', background='yellow',
                       anchor=tkinter.SE, font=('Helvetica', '24', 'bold italic'),
                       padx=10, pady=10)
label.pack(side=tkinter.LEFT, fill=tkinter.BOTH, expand=True)
# parent window
close_button = tkinter.Button(root, text='Close', command=do_quit)
close_button.pack(side=tkinter.RIGHT)
tkinter.mainloop() # loop until all windows are closed/destroyed
```

Welcome example (class)

welcome_class.py

```
import tkinter

class Welcome:
    def do_quit(self): # event handler for 'Close'
        self.root.destroy()

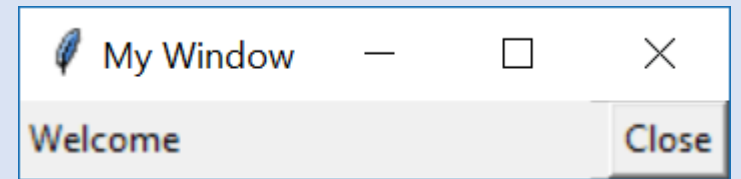
    def __init__(self, window_title):
        self.root = tkinter.Tk()
        self.root.title(window_title)

        self.label = tkinter.Label(self.root, text='Welcome')
        self.label.pack(side=tkinter.LEFT)

        self.close_button = tkinter.Button(self.root, text='Close', command=self.do_quit)
        self.close_button.pack(side=tkinter.RIGHT)

Welcome('My Window')

tkinter.mainloop()
```



increment.py (part I)

```
import tkinter

class Counter:
    def do_quit(self):
        self.root.destroy()

    def add(self, x):
        self.counter += x
        self.count.set(self.counter)

    def __init__(self, message):
        self.counter = 0

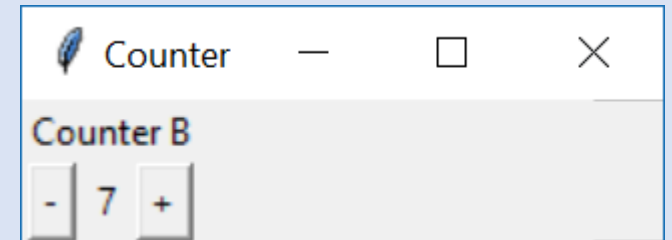
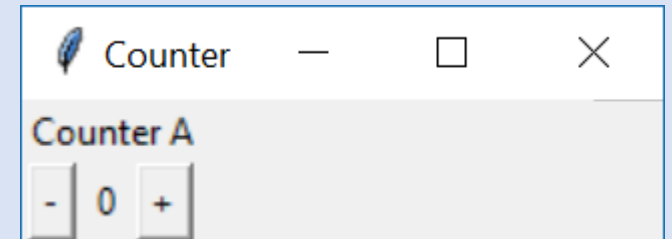
        self.root = tkinter.Toplevel() # new window
        self.root.title('Counter')

        self.label = tkinter.Label(self.root, text=message)
        self.label.grid(row=0, columnspan=3)

        self.minus_button = tkinter.Button(self.root, text='-', command=lambda: self.add(-1))
        self.minus_button.grid(row=1, column=0)

        self.count = tkinter.IntVar()
        self.count_label = tkinter.Label(self.root, textvariable=self.count)
        self.count_label.grid(row=1, column=1)

        self.plus_button = tkinter.Button(self.root, text='+', command=lambda: self.add(+1))
        self.plus_button.grid(row=1, column=2)
```



increment.py (part II)

```
class Counter_app:
    def __init__(self):
        self.counters = 0

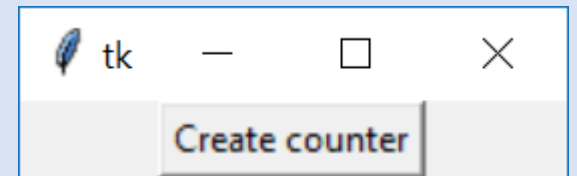
        self.root = tkinter.Tk()

        self.create = tkinter.Button(self.root, text='Create counter', command=self.new_counter)
        self.create.pack()

    def new_counter(self):
        Counter('Counter ' + chr(ord('A') + self.counters))
        self.counters += 1

Counter_app()

tkinter.mainloop()
```



Canvas

`canvas.py`

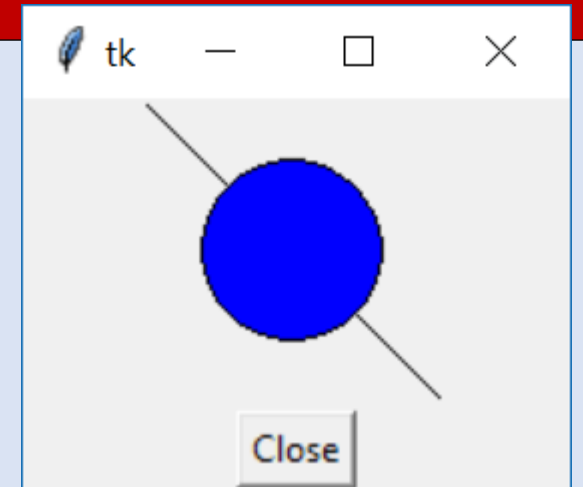
```
import tkinter

root = tkinter.Tk()

canvas = tkinter.Canvas(root, width=100, height=100)
canvas.pack()
canvas.create_line(0, 0, 100, 100)
canvas.create_oval(20, 20, 80, 80, fill='blue')

close = tkinter.Button(root, text='Close', command=root.destroy)
close.pack()

tkinter.mainloop()
```





Calculator



42

7

8

9

*

C

4

5

6

/

%

1

2

3

-

=

0

.

+

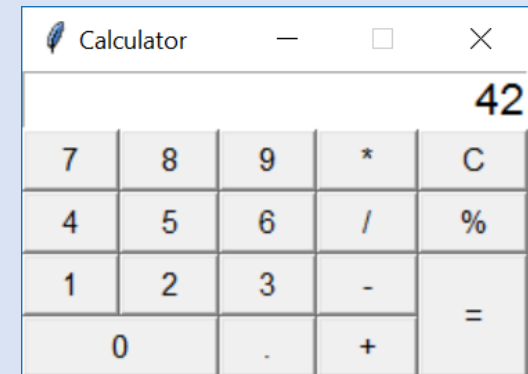
calculator.py (Part I)

```
import tkinter
from tkinter import messagebox

class Calculator:
    def __init__(self, root):
        self.root = root

        self.display = tkinter.Entry(self.root, font=('Helvetica', 16), justify=tkinter.RIGHT)
        self.display.insert(0, '0')
        self.display.grid(row=0, column=0, columnspan=5) # grid = geometry manager

        self.button(1, 0, '7')
        self.button(1, 1, '8')
        self.button(1, 2, '9')
        self.button(1, 3, '*')
        self.button(1, 4, 'C', command=self.clearText) # 'C' button
        self.button(2, 0, '4')
        self.button(2, 1, '5')
        self.button(2, 2, '6')
        self.button(2, 3, '/')
        self.button(2, 4, '%')
        self.button(3, 0, '1')
        self.button(3, 1, '2')
        self.button(3, 2, '3')
        self.button(3, 3, '-')
        self.button(3, 4, '=', rowspan=2, command=self.calculateExpression) # '=' button
        self.button(4, 0, '0', columnspan=2)
        self.button(4, 2, '.')
        self.button(4, 3, '+')
```



calculator.py (Part II)

```
def button(self, row, column, text, command=None, columnspan=1, rowspan=1):
    if command == None:
        command = lambda: self.appendToDisplay(text)
    B = tkinter.Button(self.root, font=('Helvetica', 11), text=text, command=command)
    B.grid(row=row, column=column, rowspan=rowspan, columnspan=columnspan, sticky='NWESWSE')

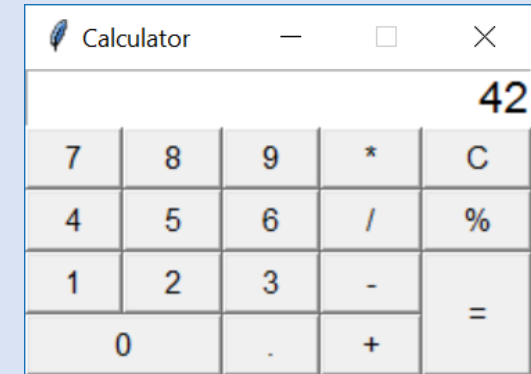
def clearText(self):
    self.replaceText('0')

def replaceText(self, text):
    self.display.delete(0, tkinter.END)
    self.display.insert(0, text)

def appendToDisplay(self, text):
    if self.display.get() == '0':
        self.replaceText(text)
    else:
        self.display.insert(tkinter.END, text)

def calculateExpression(self):
    expression = self.display.get().replace('%', '/ 100')
    try:
        result = eval(expression) # DON'T DO THIS !!!
        self.replaceText(result)
    except:
        messagebox.showwarning('Message', 'Invalid expression')

root = tkinter.Tk()
root.title('Calculator')
root.resizable(0, 0) # disallow resizing and maximizing the window
Calculator(root)
tkinter.mainloop()
```



Creating a menu

rectangles.py

```
class Rectangles:
    Colors = ['black', 'red', 'blue', 'green', 'yellow']

    def create_menu(self):
        menubar = tkinter.Menu(self.root)
        menubar.add_command(label='Quit! (Ctrl-q)', command=self.do_quit)

        editmenu = tkinter.Menu(menubar, tearoff=0)
        editmenu.add_command(label='Clear', command=self.clear_all)
        editmenu.add_command(label='Delete last (Ctrl-z)', command=self.delete_last_rectangle)

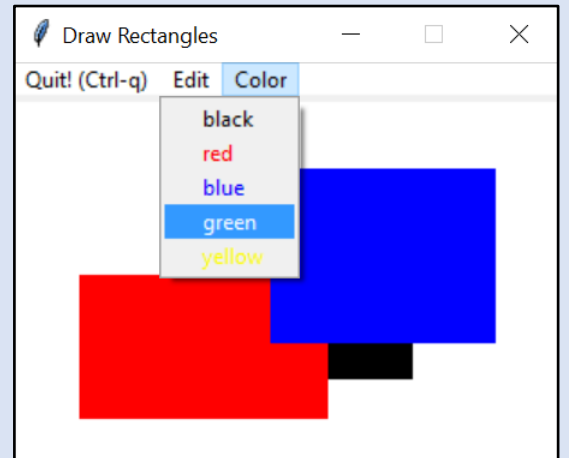
        colormenu = tkinter.Menu(menubar, tearoff=0)
        for color in self.Colors: # list of color names
            colormenu.add_command(label=color,
                                   foreground=color,
                                   command=self.get_color_handler(color))

        menubar.add_cascade(label='Edit', menu=editmenu)
        menubar.add_cascade(label='Color', menu=colormenu)
        self.root.config(menu=menubar) # Show menubar

    def get_color_handler(self, color):
        return lambda : self.set_color(color)

    def set_color(self, color):
        self.current_color = color

    ...
```

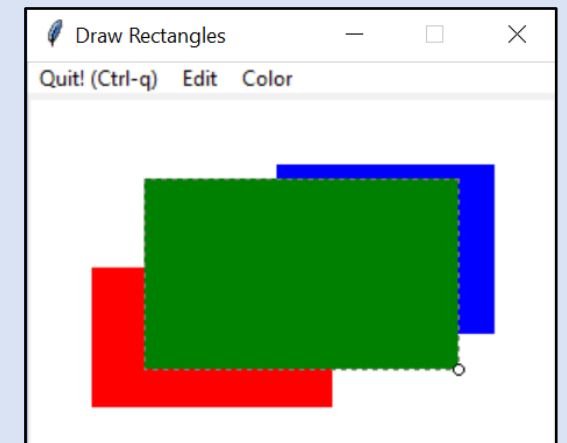


Binding key and mouse events

- Whenever a key is pressed, mouse button is pressed/released, mouse is moved, mouse enters/leaves objects etc. **events** are triggered that can be bound to call a user defined **event handler**

rectangles.py (continued)

```
...
self.root = tkinter.Tk()
self.root.bind('<Control-q>', self.do_quit)
self.root.bind('<Control-z>', self.delete_last_rectangle)
...
self.canvas = tkinter.Canvas(self.root, width=300, height=200,
                              background='white')
self.canvas.bind('<Button-1>', self.create_rectangle_start)
self.canvas.bind('<B1-Motion>', self.create_rectangle_mouse_move)
self.canvas.bind('<ButtonRelease-1>', self.create_rectangle_end)
...
```



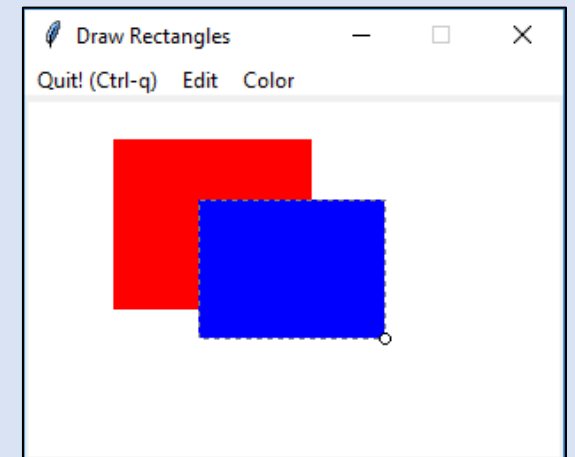
Handling mouse events

rectangles.py (continued)

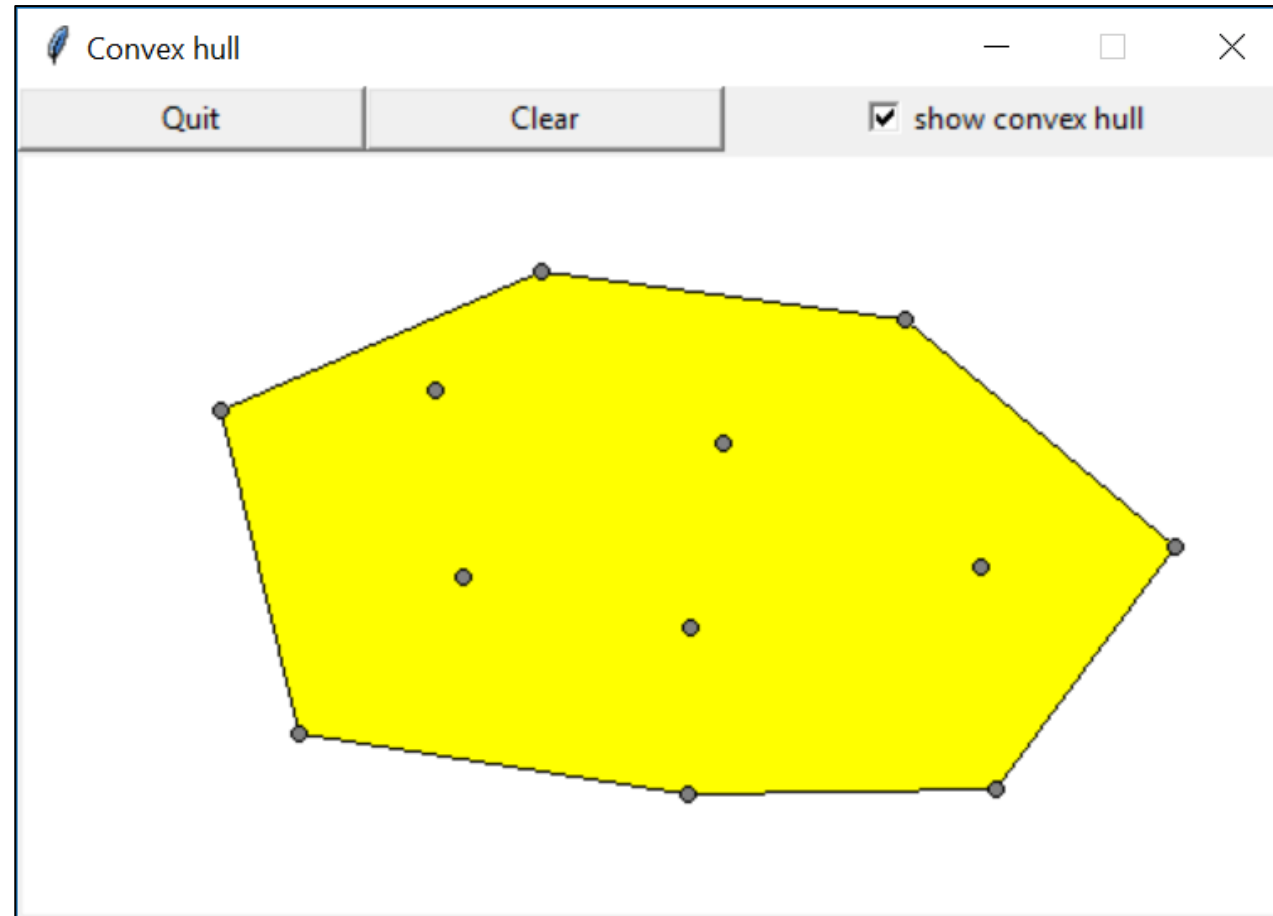
```
def create_rectangle_start(self, event):
    radius = 3
    x, y = event.x, event.y
    self.top_pos = (x, y)
    self.bottom_pos = (x, y)
    self.rectangle = self.canvas.create_rectangle(x, y, x, y, # top-left = bottom-right
                                                  fill=self.current_color, width=1, outline='grey', dash=(3, 5))
    self.corner = self.canvas.create_oval(x - radius, y - radius, x + radius, y + radius, fill='white')

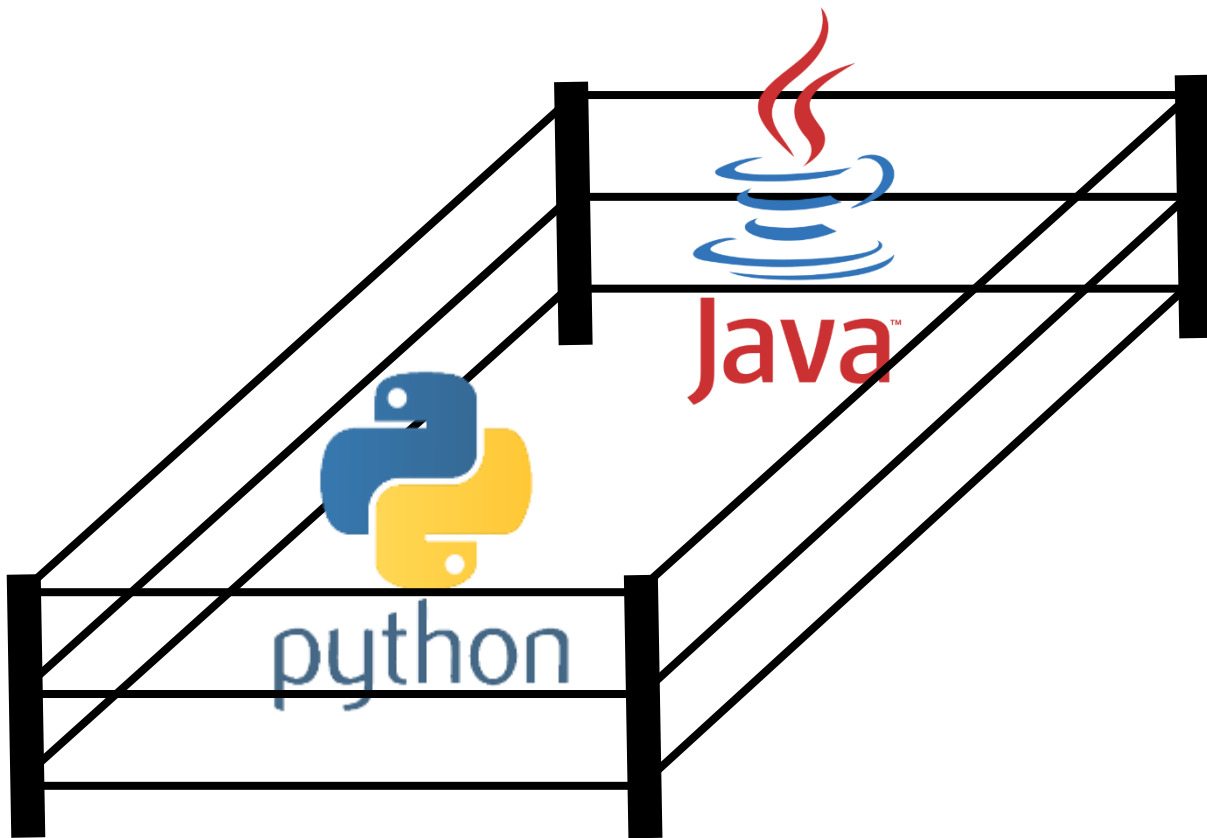
def create_rectangle_mouse_move(self, event):
    if self.corner:
        x, y = event.x, event.y
        x_, y_ = self.bottom_pos
        self.bottom_pos = (x, y)
        self.canvas.coords(self.rectangle, *self.top_pos, *self.bottom_pos)
        self.canvas.move(self.corner, x - x_, y - y_)

def create_rectangle_end(self, event):
    if self.corner:
        self.canvas.delete(self.corner)
        self.corner = None
    if self.bottom_pos != self.top_pos:
        self.rectangles.append(self.rectangle)
        self.canvas.itemconfig(self.rectangle, width=0)
    else: # empty rectangle, skip
        self.canvas.delete(self.rectangle)
    self.rectangle = None
```



Exercise 25.1 (convex hull GUI)

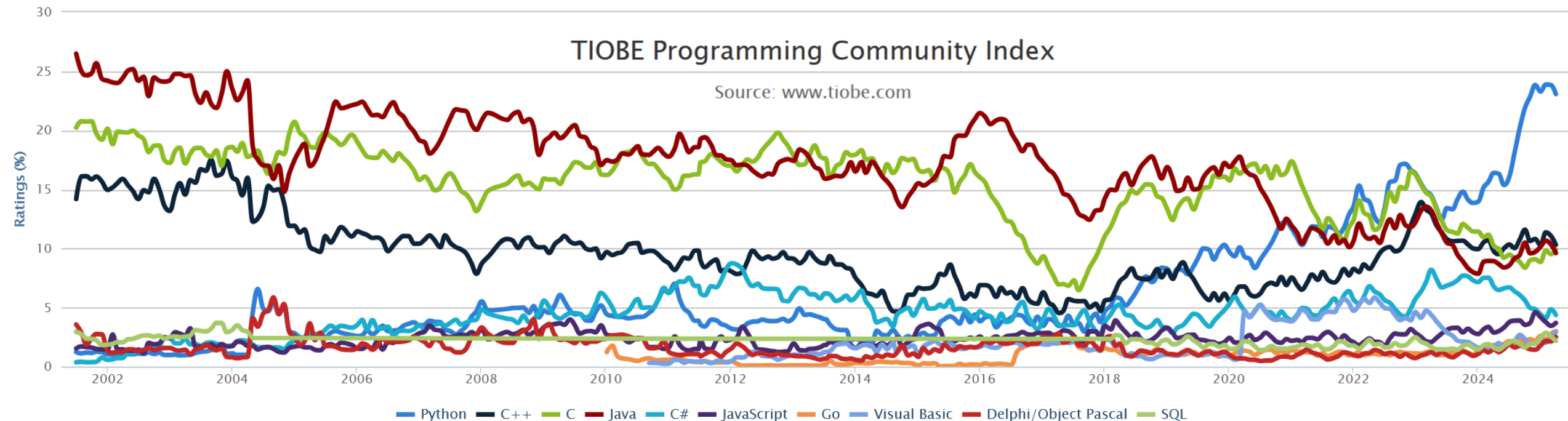




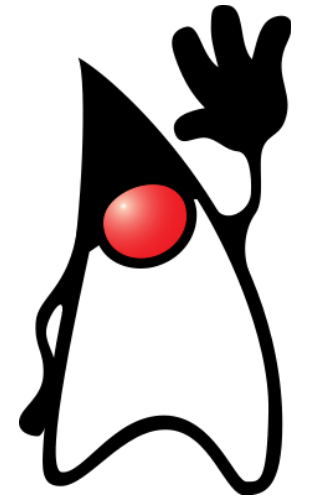
Java vs Python

Why should you know something about Java?

- Java is an example of a **statically typed object oriented** language (like C and C++) opposed to Python's being dynamically typed
- One of the most widespread used programming languages
- Used in other courses at the Department of Computer Science



Java history



- Java 1.0 released 1995 by Sun Microsystems (acquired by Oracle 2010)
- "Write Once, Run Anywhere"
- 1999 improved performance by the **Java HotSpot Performance Engine**
- Current version Java 24 (released March 2025)
- Java compiler generates **Java bytecode** that is executed on a **Java virtual machine (JVM)** that contains a garbage collector

PyPy is adopting the same ideas to Python (Just-in-Time compilation)

Installing Java

- To compile Java programs into **bytecode** you need a **compiler**, e.g. from Java SE Development Kit (**JDK**):

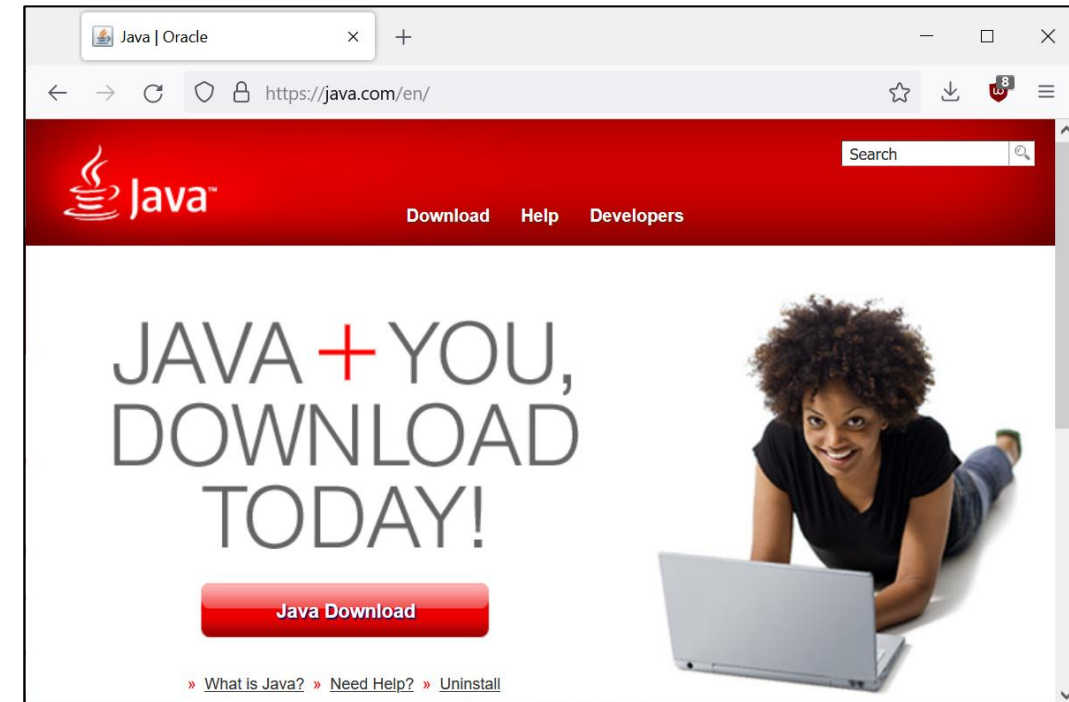
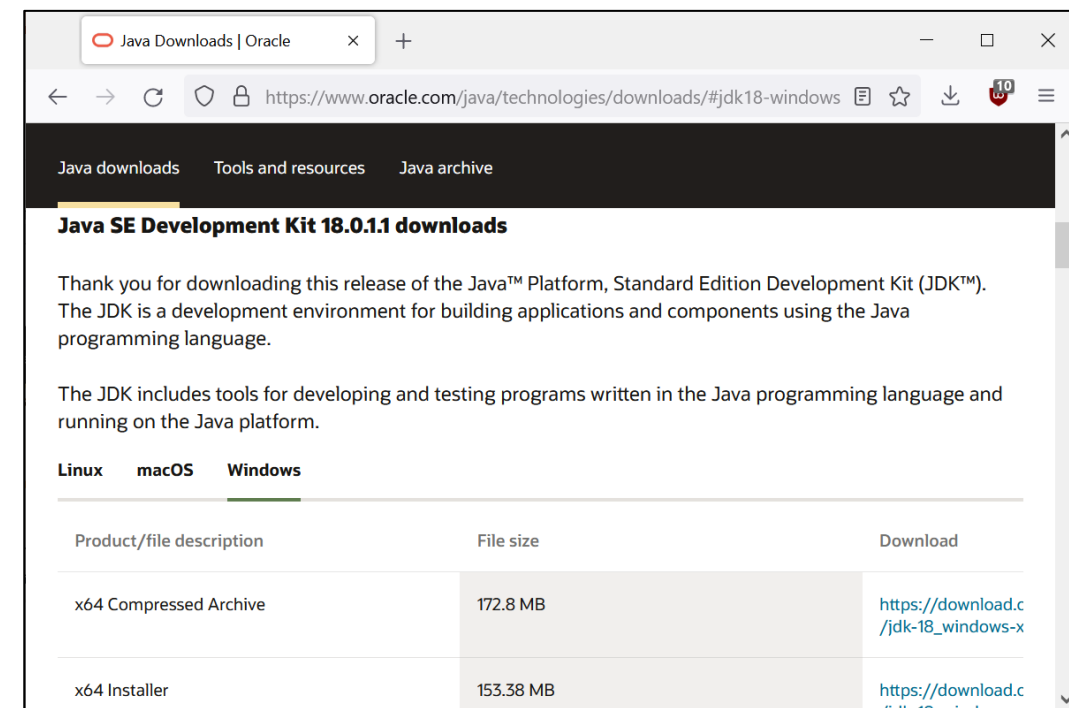
www.oracle.com/java/technologies/downloads/

(you might need to add the JDK directory to your PATH, e.g. C:\Program Files\Java\jdk-18.0.1.1\bin)

- To only run compiled Java programs:

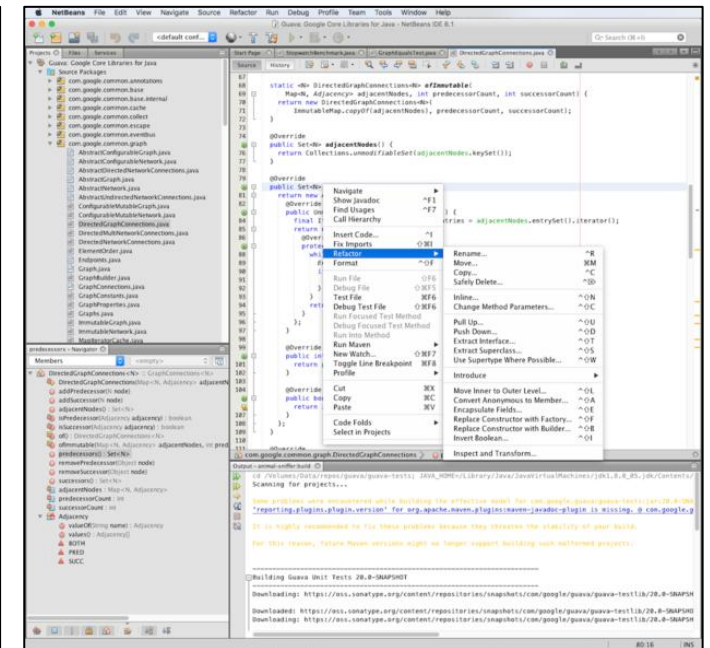
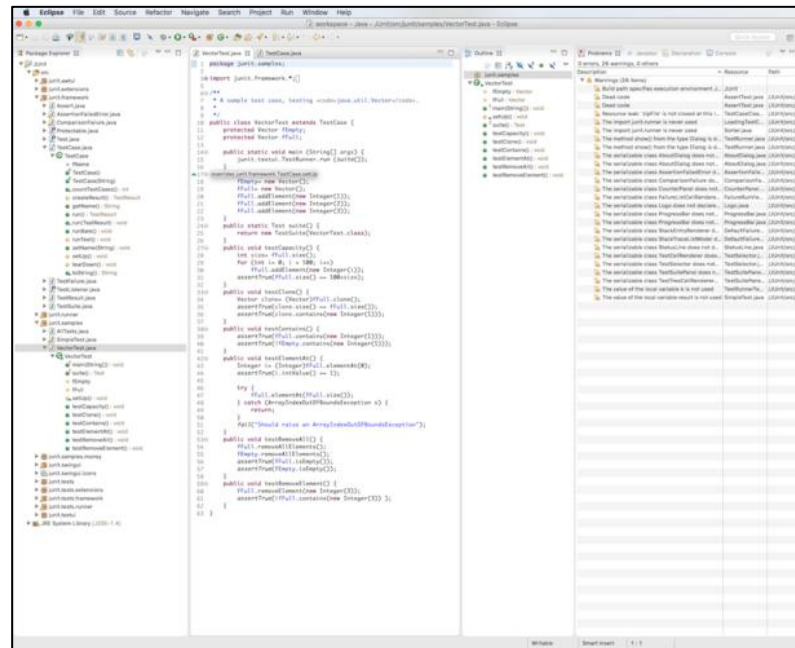
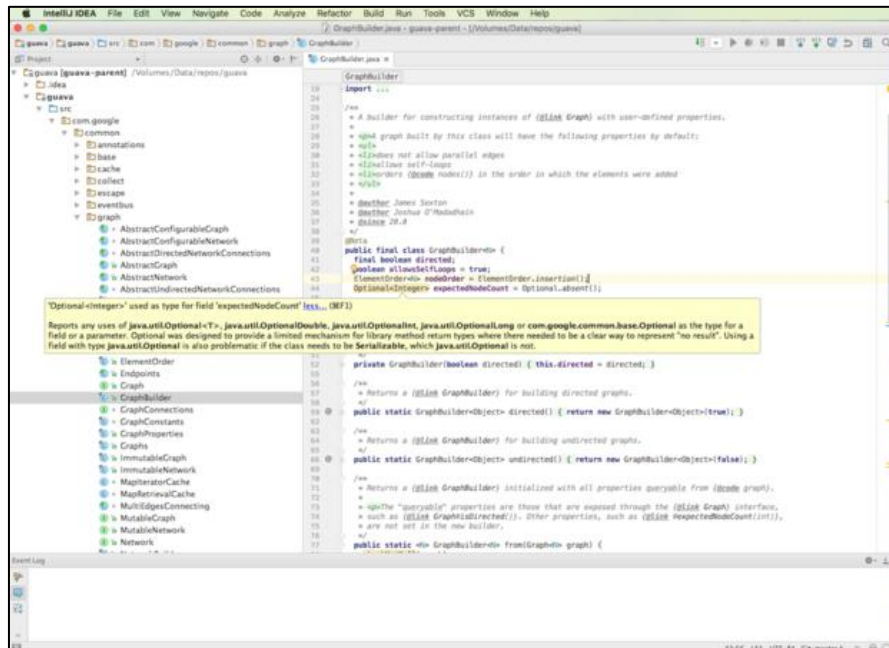
java.com/download

(If you use JDK, you should not download this)

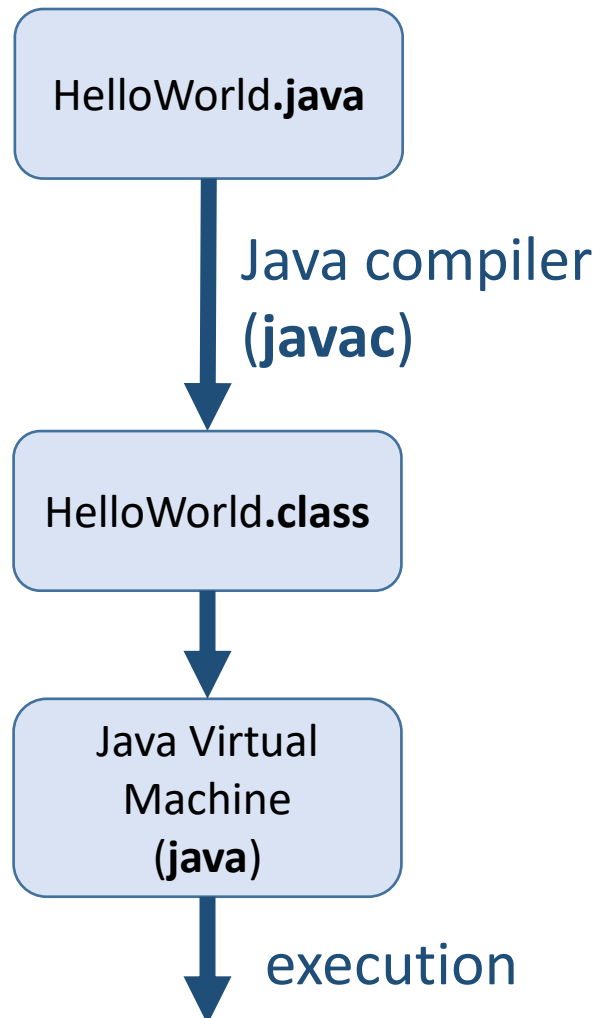


Java IDE

- Many available, some popular:
Visual Studio Code, IntelliJ IDEA, Eclipse, and NetBeans
- An IDE for beginners: BlueJ



Compiling and running a Java program



HelloWorld.java

```
public class HelloWorld {  
    public static void main( String[] args ) {  
        System.out.println( "Hello World!" );  
    }  
}
```

Command Prompt

C:\Users\au121\Desktop>javac HelloWorld.java

C:\Users\au121\Desktop>dir HelloWorld*

Volume in drive C is OSDisk
Volume Serial Number is 3CDB-90D8

Directory of C:\Users\au121\Desktop

04-05-2020	17:40	426	HelloWorld.class
09-05-2019	21:18	132	HelloWorld.java
		2 File(s)	558 bytes
		0 Dir(s)	372.191.944.704 bytes free

C:\Users\au121\Desktop>java HelloWorld

Hello World!

C:\Users\au121\Desktop>

Java : `main`

- `name.java` must be equal to the public class `name`
- A class can only be executed using `java name` (without `.class`) if the class has a class method `main` with signature

```
public static void main(String[] args)
```

- (`main` is inherited from C and C++ sharing a lot of syntax with Java)
- Java convention is that class names should use CamelCase

`PrintArguments.java`

```
public class PrintArguments {  
    public static void main( String[] args ) {  
        for (int i=0; i<args.length; i++)  
            System.out.println( args[i] );  
    }  
}
```

`shell`

```
> java PrintArguments x y z  
| x  
| y  
| z
```


a static method in
a class is a class method
(exists without creating objects)

method name

type of return value
(void = no return value)

class name containing main
must have same name as file

type of argument,
array of String values

there can be several classes
in a file – but only one class
should be public and have
same name as file

the main method must
be public to be visible
outside class

PrintArguments.java

```
public class PrintArguments {  
    public static void main( String[] args ) {  
        for (int i=0; i<args.length; i++)  
            System.out.println( args[i] );  
    }  
}
```

name of argument

For-loop equivalent to

```
int i = 0;  
while (i < args.length) {  
    code  
    i++;  
}
```

i += 1

declare new int variable
locally inside for-loop

the print statement is found
in the System class

java arrays are indexed from
0 to args.length - 1
and the length of an array
object is fixed once created

Argument list also exists in Python...

```
PrintArguments.py
```

```
import sys  
print(sys.argv)
```

```
shell
```

```
> python PrintArguments.py a b 42  
| ['PrintArguments.py', 'a', 'b', '42']
```

Primitive.java

```
/**
 * A Java docstring to be processed using 'javadoc'
 */
// comment until end-of-line
public class Primitive {
    public static void main( String[] args ) {
        int x; // type of variable must be declared before used
        x = 1; // remember ';' after each statement
        int y=2; // indentation does not matter
        int a=3, b=4; // multiple declarations and initialization
        System.out.println(x + y + a + b);
        int[] v={1, 2, 42, 3}; // array of four int
        System.out.println(v[2]); // prints 42, arrays 0-indexed
        /* multi-line comment
           that continues until here */
        v = new int[3]; // new array of size three, containing zeros
        System.out.println(v[2]); // prints 0
        if (x == y) { // if-syntax '(' and ')' mandatory
            a = 1;
            b = 2;
        } else { // use '{' and '}' to create block of statements
            a = 4; b = 3; // two statements on one line
        }
    }
}
```

Why state types – Python works without...

- Just enforcing a different programming style (also C and C++)
- Helps users to avoid mixing up values of different types
- (Some) type errors can be caught at compile time
- More efficient code execution

type_error.py

```
x = 3
y = 'abc'
print('program running...')
print(x / y)
```

Python shell

```
| program running...
...
| ----> 4 print(x / y)
| TypeError: unsupported operand type(s) for
| /: 'int' and 'str'
```

TypeError.java

```
public class TypeError {
    public static void main( String[] args ) {
        int x = 3;
        String y = "abc";
        System.out.println(x / y);
    }
}
```

shell

```
> javac TypeError.java
| javac TypeError.java
| TypeError.java:5: error: bad operand types for
| binary operator '/'
|         System.out.println(x / y);
|                             ^
|         first type:  int
|         second type: String
| 1 error
```

Basic Java types

Type	Values
boolean	true or false
byte	8 bit integer
char	character (16-bit UTF)
short	16 bit integer
int	32 bit integer
long	64 bit integer
float	32 bit floating bout
double	64 bit floating point
class BigInteger	arbitrary precision integers
class String	strings

BigIntegerTest.java

```
import java.math.*; // import everything
// import java.math.BigInteger; // alternativ

public class BigIntegerTest {
    public static void main( String[] args ) {
        BigInteger x = new BigInteger("2");
        while (true) {
            // BigIntegers are immutable
            x = x.multiply(x);
            // java.math.BigInteger.toString()
            System.out.println(x);
        }
    }
}
```

shell

```
4
16
256
65536
4294967296
18446744073709551616
340282366920938463463374607431768211456
...
```

Java arrays

- The size of a builtin Java array can not be modified when first created. If you need a bigger array you have to instantiate a new array.
- Or better use a standard collection class like **ArrayList**
- **ArrayList** is a generic class (type of content is given by *<element type>*; generics available since Java 5, 2004)
- The for-each loop was introduced in Java 5

ConcatenateArrayLists.java

```
import java.util.*; // java.util contains ArrayList

public class ConcatenateArrayList {
    public static void main( String[] args ) {
        // ArrayList is a generic container
        ArrayList<String> a = new ArrayList<String>();
        ArrayList<String> b = new ArrayList<String>();
        ArrayList<String> c = new ArrayList<String>();

        a.add("A1"); // in Python .append
        a.add("A2");
        b.add("B1");

        c.addAll(a); // in Python .extend
        c.addAll(b);

        for (String e : c) { // foreach over iterator
            System.out.println(e);
        }
    }
}
```

shell

```
| A1
| A2
| B1
```

Tired of writing all these types...

- In Java 7 (2011) the “diamond operator” `<>` was introduced for type inference for generic instance creation to reduce verbosity
- In Java 10 (2018) the `var` keyword was introduced to type infer variables

ArrayListTest.java

```
import java.util.*; // java.util contains ArrayList

public class ArrayListTest {
    public static void main( String[] args ) {
        // ArrayList is a generic container
        ArrayList<String> a = new ArrayList<String>(); // Full types
        List<String> b = new ArrayList<String>();      // ArrayList is subclass of class List
        ArrayList<String> c = new ArrayList<>();       // <> uses type inference
        List<String> d = new ArrayList<>();            // <> and ArrayList subclass of List
        var e = new ArrayList<String>();               // use var to infer type of variable
         var v = Math.floor(1.5); // not obvious what type v is (double)
    }
}
```

Function arguments

- Must declare the number of arguments and their types, and the return type
- The argument types are part of the *signature* of the function
- Several functions can have the same name, but different type signatures
- Python keyword arguments, * and ** do not exist in Java 😞

Functions.java

```
public class Functions {  
    private static int f(int x) {  
        return x * x;  
    }  
    private static int f(int x, int y) {  
        return x * y;  
    }  
    private static String f(String a, String b) {  
        return a + b; // string concatenation  
    }  
  
    public static void main( String[] args ) {  
        System.out.println(f(7));  
        System.out.println(f(3, 4));  
        System.out.println(f("abc", "def"));  
    }  
}
```

shell

```
| 49  
| 12  
| abcdef
```

functions.py

```
def f(x, y=None):  
    if y == None:  
        y = x  
    if type(x) is int:  
        return x * y  
    else:  
        return x + y  
print(f(7), f(3, 4), f('abc', 'def'))
```


Class

- Constructor = method with name equal to class name (no return type)
- `this` = refers to current object (Python “self”)
- Use `private` / `public` on attributes / methods to give access outside class
- Use `new` *name*(arguments) to create new objects
- There can be multiple constructors, but with distinct type signatures

AClass.java

```
class Rectangle {
    private int width, height; // declare attributes
    // constructor, class name, no return type
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public Rectangle(int side) {
        width = side; // same as this.width = side
        height = side;
    }

    public int area() {
        return width * height;
    }
}

public class AClass {
    public static void main( String[] args ) {
        Rectangle r = new Rectangle(6, 7);
        System.out.println(r.area());
    }
}
```

shell

Inheritance

- Java supports single inheritance using `extends`
- Attributes and methods that should be accessible in a subclass must be declared `protected` (or `public`)
- Constructors are not inherited but can be called using `super`

Inheritance.java

```
class BasicRectangle {
    // protected allows subclass to access attributes
    protected int width, height;
    public BasicRectangle(int width, int height) {
        this.width = width; this.height = height;
    }
}

class Rectangle extends BasicRectangle {
    public Rectangle(int width, int height) {
        // call constructor of super class
        super(width, height);
    }
    public int area() {
        return width * height;
    }
}

public class Inheritance {
    public static void main( String[] args ) {
        Rectangle r = new Rectangle(6, 7);
        System.out.println(r.area());
    }
}
```

shell

Generic class

- Class that is parameterized by one or more types (comma separated)
- Primitive types cannot be type parameters
- Instead use *wrappers*, like **Integer** for **int**

GenericPair.java

```
class Pair<element> {  
    private element x, y;  
    public Pair(element x, element y) {  
        this.x = x; this.y = y;  
    }  
    element first() { return x; }  
    element second() { return y; }  
}  
  
public class GenericPair {  
    public static void main( String[] args ) {  
        var p = new Pair<Integer>(6, 7);  
        System.out.println(p.first() * p.second());  
    }  
}
```

shell

Interface

- Java does not support multiple inheritance like Python
- But a class can implement an arbitrary number of **interfaces**
- An interface specifies a set of attributes and methods a class must have
- The type of a variable can be an interface, and the variable can hold any object where the class is stated to **implement** the interface

RectangleInterface.java

```
interface Shape {
    public int area(); // method declaration
}

class Rectangle implements Shape {
    private int width, height;

    // constructor, class name, no return type
    public Rectangle(int width, int height) {
        this.width = width; this.height = height;
    }

    public int area() {
        return width * height;
    }
}

public class RectangleInterface {
    public static void main( String[] args ) {
        Shape r = new Rectangle(6, 7);
        System.out.println(r.area());
    }
}
```

Abstract classes

- **Abstract class** = class that cannot be instantiated, labeled `abstract`
- **Abstract method** = method declared without definition, labeled `abstract`, must be in abstract class
- An abstract class can be **extended** to a non-abstract class by providing the missing method definitions

AbstractRectangle.java

```
abstract class Shape {
    abstract public int circumference();
    abstract public int area();
    public double fatness() {
        // convert int from area() to double before /
        return (double)area() / circumference();
    }
}

class Rectangle extends Shape {
    private int width, height;
    // constructor, class name, no return type
    public Rectangle(int width, int height) {
        this.width = width; this.height = height;
    }
    public int area() {
        return width * height;
    }
    public int circumference() {
        return 2 * (width + height);
    }
}

public class AbstractRectangle {
    public static void main( String[] args ) {
        Shape r = new Rectangle(6, 7);
        System.out.println(r.fatness());
    }
}
```

Default methods in interfaces

- Before Java 8 all methods in an interface were abstract (no definition)
- Since Java 8 interfaces can have **default** methods with definition
- The distinction between abstract classes and interfaces gets blurred
 - a class can only extend one abstract class
 - a class can implement more interfaces⇒ multiple “inheritance” is possible in Java

DefaultInterface.java

```
interface Shape {
    public int circumference();
    public int area();
    default public double fatness() {
        // convert int from area() to double before /
        return (double)area() / circumference();
    }
}

class Rectangle implements Shape {
    private int width, height;
    // constructor, class name, no return type
    public Rectangle(int width, int height) {
        this.width = width; this.height = height;
    }
    public int area() {
        return width * height;
    }
    public int circumference() {
        return 2 * (width + height);
    }
}

public class DefaultRectangle {
    public static void main( String[] args ) {
        Shape r = new Rectangle(6, 7);
        System.out.println(r.fatness());
    }
}
```

Multiple Inheritance

- Class C implements both interfaces A and B
- Inherits default methods sayA and sayB
- Cannot inherit sayHi, since in both A and B. Must be overridden in C
- Can use @override to enforce compiler to check if method exists in super class
- new A() {} creates an instance of an *anonymous class* (extending or implementing A)

MultipleInheritance.java

```
interface A {
    default public void sayA() { System.out.println("say A"); }
    default public void sayHi() { System.out.println("A say's Hi"); }
}

interface B {
    default public void sayB() { System.out.println("say B"); }
    default public void sayHi() { System.out.println("B say's Hi"); }
}

class C implements A, B {
    @Override // (optional) requests compiler to
               // check if sayHi exists in supertype
    public void sayHi() {
        System.out.println("C say's Hi");
        (new A() {}).sayHi(); // instantiate an anonymous class
    }

    public void test() {
        sayA();
        sayB();
        sayHi();
    }
}

public class MultipleInheritance {
    public static void main( String[] args ) {
        new C().test();
    }
}
```

Shell output

```
say A
say B
C say's Hi
A say's Hi
```

Lambda expression

- Lambda expressions are possible since Java 8
- Syntax : *argument -> expression*

LambdaPrinting.java

```
import java.util.*; // ArrayList
public class LambdaPrinting {
    public static void main(String[] args) {
        var elements = new ArrayList<Integer>();
        for (int i = 1; i <= 3; i++)
            elements.add(i);
        elements.forEach(e -> System.out.println(e));
    }
}
```

LambdaPrinting.java

```
1
2
3
```




Welcome to Java for Python Programmers

Contents:

- 1. Java for Python Programmers
 - 1.1. Preface
 - 1.2. Introduction
 - 1.3. Why Learn another programming Language?
 - 1.3.1. Why Learn Java? Why not C or C++?
 - 1.4. Lets look at a Java Program
 - 1.5. Java Data Types
 - 1.5.1. Numeric
 - 1.5.1.1. Import
 - 1.5.1.2. Declaring Variables
 - 1.5.1.3. Input / Output / Scanner
 - 1.5.2. String
 - 1.5.3. List
 - 1.5.4. Arrays



Introduction to Programming with Scientific Applications

- Missing handins, deadline 16 May 2025
- Final project, deadline 31 May 2025
- Course evaluation
- Exam, 14 June 2025
- AOB

Exam

- 5 hours, written exam, with aids, including PC and internet
- Communication with others about the exam is not permitted during the exam
- WISEflow Device Monitor must be enabled
- AI assistants like ChatGPT and GitHub Copilot are not allowed
- Reexam in August
- Grade is an *overall assessment* of the implementation project and the exam
 - The result of the final exam must meet the minimum requirements for acceptance to be able to pass the course
 - The final exam contributes 80% to the final grade
- wiseflow.au.dk
 - Download .zip + add missing code + upload .zip
- Questions? – post them on Brightspace

Content of .zip file

```
A.py - C:\Users\au121\Desktop\ipsa22\ipsa22exam\A.py (3.11.3)
File Edit Format Run Options Window Help

INTERVAL SUM

Your task is to write a function interval_sum(i, j), that returns
the sum i + (i + 1) + ... + j. Eg. for i = 10 and j = 13 the sum
returned should be 10 + 11 + 12 + 13 = 46.

Input: Two lines, containing integers i and j, respectively.
It is guaranteed that 1 <= i <= j <= 100.

Output: The sum i + (i + 1) + ... + j.

Example:

Input:  10
       13

Output: 46

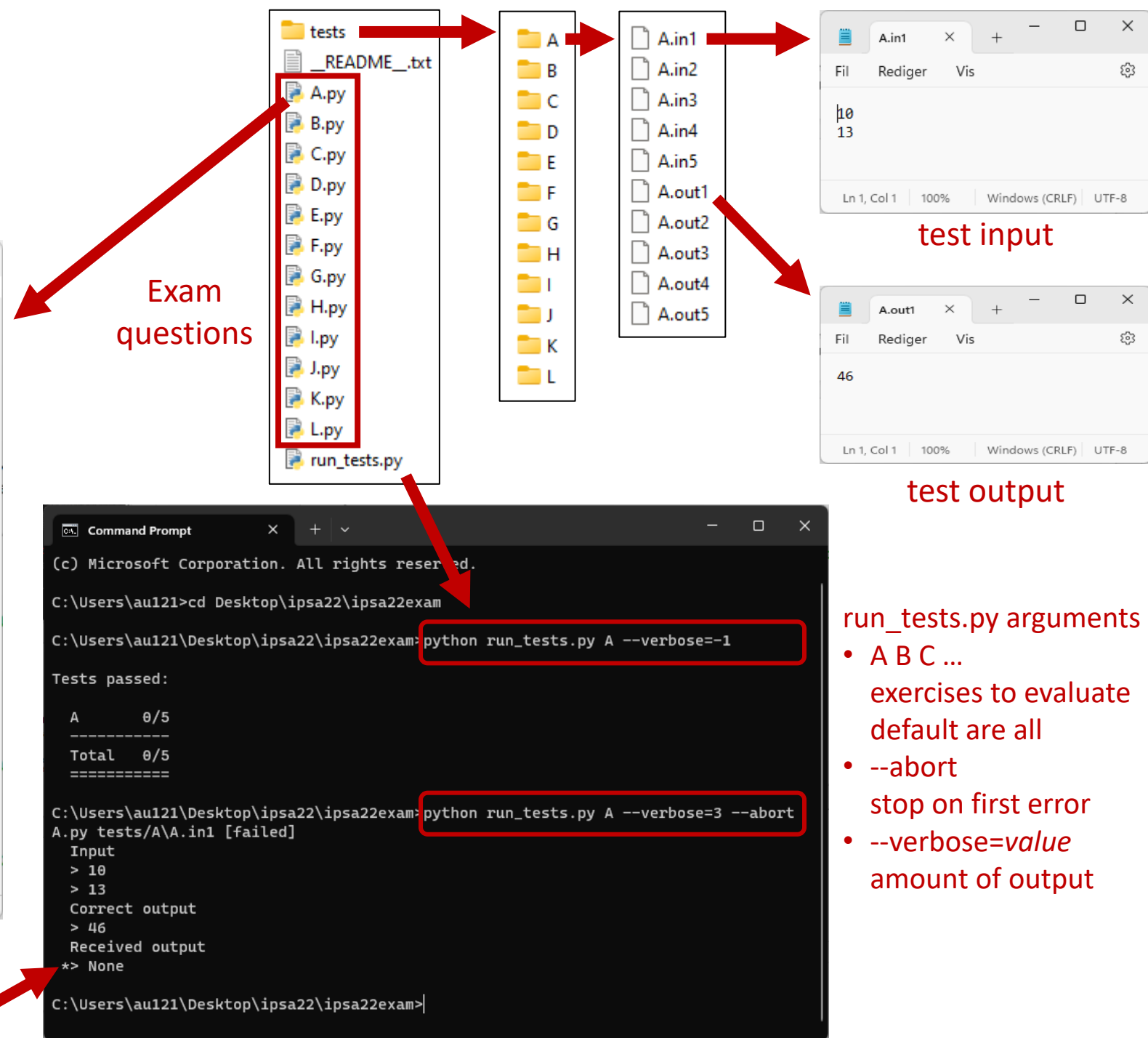
Note: The below code already handles the input and output.
...

def interval_sum(i, j):
    # insert code
    pass

i = int(input())
j = int(input())
print(interval_sum(i, j))

Ln: 1 Col: 0
```

* = line with wrong output



Evaluation of code

Don't expect partial scores
for this solution

```
def interval_sum(i, j):  
    if i == 10 and j == 13:  
        return 46
```

- Each problem will be assigned a **weight**
- There will be problems of **varying difficulty**
- Code will be evaluated on **known test cases** and **unknown test cases**
- In general, **automatic scoring**, in some exceptional cases manual
- Googling / stack overflow / Python documentation etc. **is allowed**, but put a **comment if you copied code from internet** to avoid plagiarism

AOB ?