ANALYSIS AND MACHINE LEARNING TECHNIQUES WITH BASICS OF COMPUTER SCIENCE AND STATISTICAL LEARNING - ANALYSIS AND MACHINE LEARNING TECHNIQUES WITH BASICS OF COMPUTER SCIENCE AND STATISTICAL LEARNING

Prof. Loriano Storchi
loriano@storchi.org
https://www.storchi.org/

Contents

- Python
- Python data structures and more

Programming Laguanges

- We have seen that the programming languages may be classified as (not only but...):
  - Declarative
    - Logic
    - Functional
  - Imperative
    - Procedural
    - Object-Oriented
- Python is imperative and both procedural and object-oriented

Programming languages

- **We may classify programming languages also as dynamic typing or static typing and strong and weak e typing**
- Weak and dynamic typing, example using Perl:

```
[redo@banquo tipiz (master)]$ cat test.pl
$i = 2 + "3";
print $i, "\n";
[redo@banquo tipiz (master)]$ perl test.pl
5
[redo@banquo tipiz (master)]$
```

- *Python dynamic and strong typing*

The Python interpreter

- Python can be found ready for use on both Linux and Mac OS X. For Windows you can easily find the binaries at the following URL: http://python.org/
- Many modules (libraries) available, in our case some useful / interesting: numpy, matplotlib and pystat
- python 2.x vs python 3.x, we will use python 3.x version 2.x has been now dismissed.

The Python interpreter

```
[redo@virtualinux ~]$ python
Python 2.7.10 (default, Jun 20 2016, 14:45:40)
[GCC 5.3.1 20160406 (Red Hat 5.3.1-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> (4*5)/3
6
>>> exit()
[redo@virtualinux ~]$
```

To exit press CTRL-D or type exit()

The Python interpreter

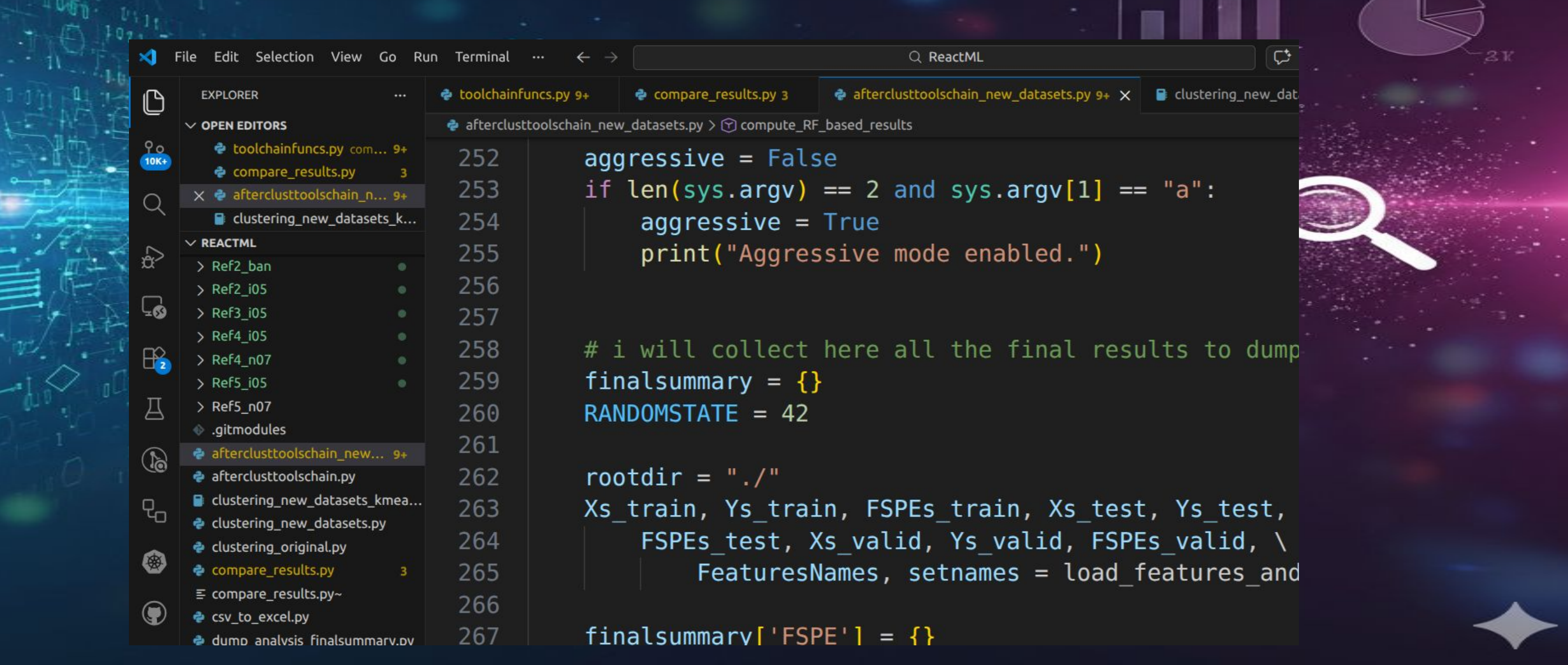
Another possibility is simply to write the source in an ASCII file and then:

$ python file.py

```
└ $ cat file.py
a = 5
b = 5

c = a/b

print(c)

|redo@buchner /home/red
└ $ python3 file.py
1.0
```

The Python interpreter

Using an IDE: VSCode



Or Jupyter

As a web application in which you can create and share documents that contain live code, equations, visualizations as well as text, the Jupyter Notebook is one of the ideal tools to help you to gain the data science skills you need.

Or Jupyter

```
[redo@buchner FondamentiDiProgrammazione]$ jupyter notebook
[I 18:32:09.367 NotebookApp] Serving notebooks from local dir
mmazione
[I 18:32:09.367 NotebookApp] The Jupyter Notebook is running
[I 18:32:09.367 NotebookApp] http://localhost:8888/?t
[I 18:32:09.367 NotebookApp] Use Control-C to stop th
[C 18:32:09.368 NotebookApp]

    Copy/paste this URL into your browser when you co
    to login with a token:
        http://localhost:8888/?token=510558c7ca812506
[I 18:32:09.617 NotebookApp] Accepting one-time-token
```

localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3

Most Visited eeegw motion RPi Cam Control v6.3....

jupyter Untitled Last Checkpoint: 3 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Run Markdown

```
In [1]: print("Hello World")

Hello World
```

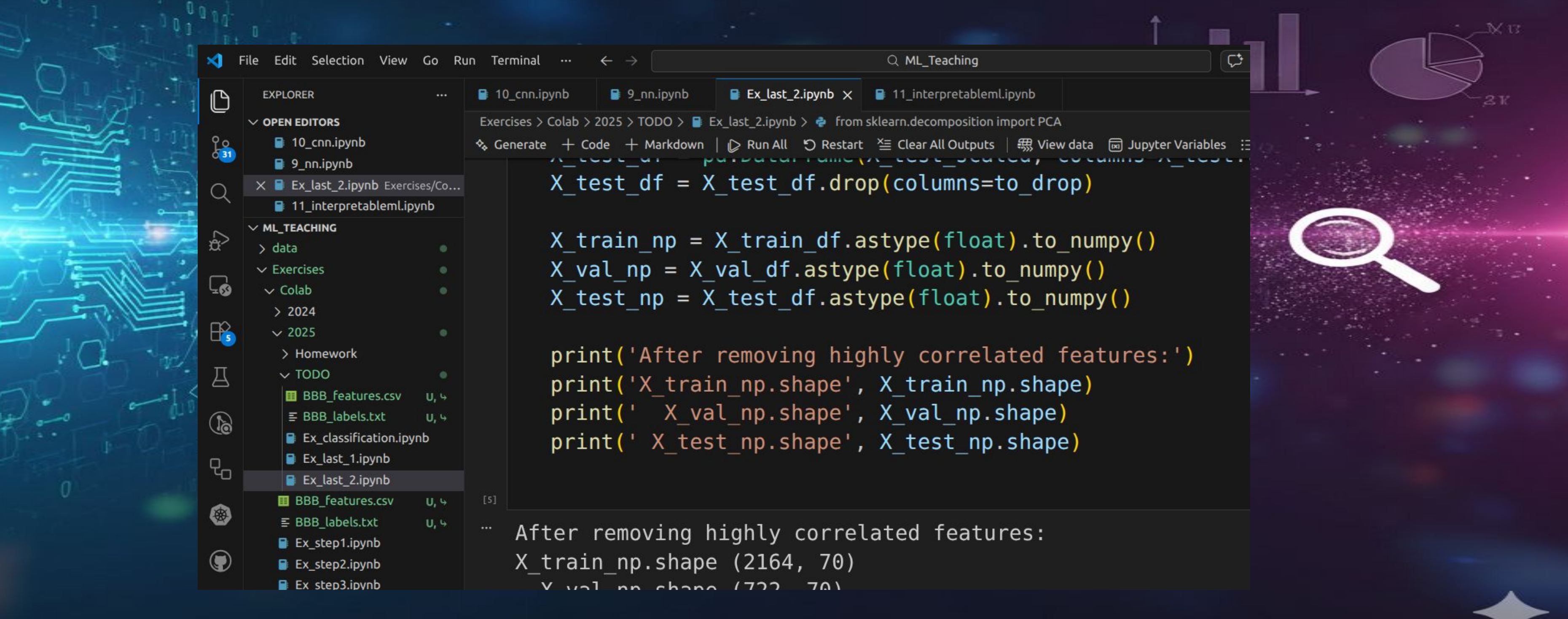
Questo e' il calssico esempio di print in python 3

```
In [2]: a = 3
        b = 4

        c = a + b

        print(c)

7
```

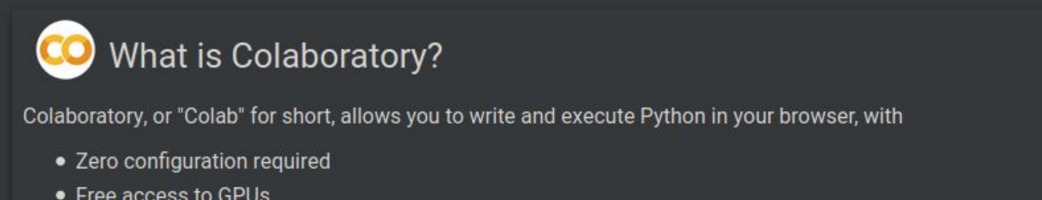
altro semplice esempio

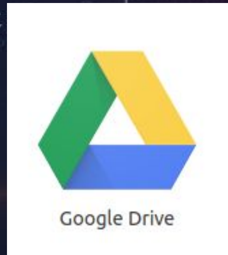
Or Jupyter

Also using an IDE: VSCode

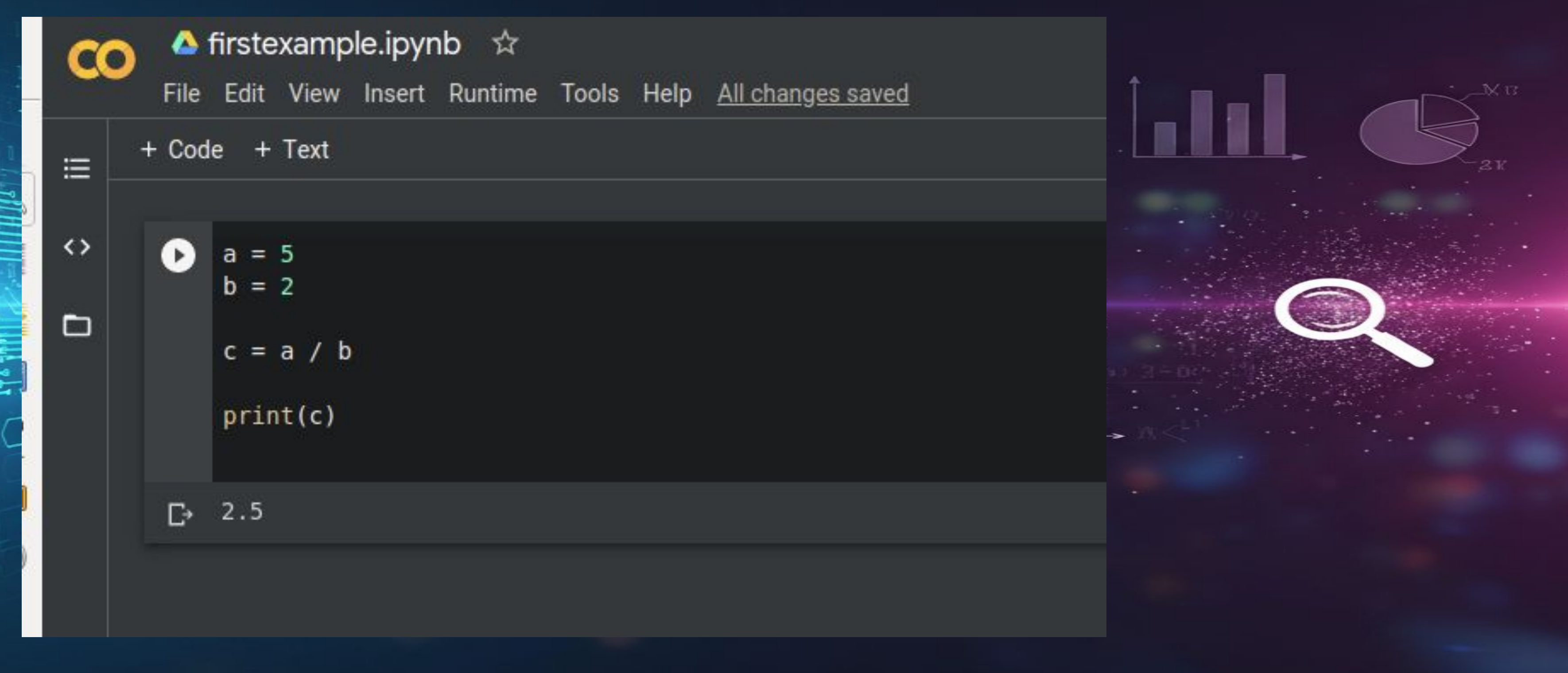


Or Colab





Or Colab



HELLO WORLD!

Hello world

- The classic program used to illustrate the syntactic basics of any programming language

```
print ("Hello World")
│redo@buchner /home/redo/L
└─ $ python3 hello.py
Hello World
```

HELLO WORLD 2

Hello world

```
└─ $ cat name.py
name = input("Insert your name: ")

print("Ciao ", name)
```

```
└─ $ python3 name.py
Insert your name: Loriano
Ciao  Loriano
```

PYTHON BASICS

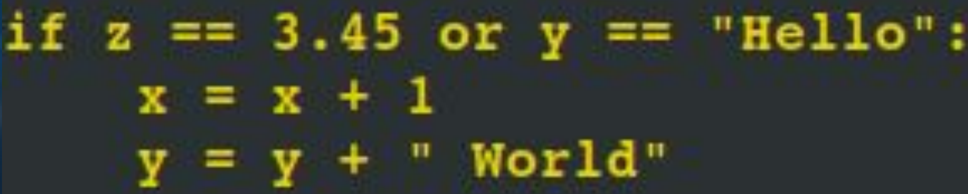
Python Basics

- Let's see a simple code that allows us to illustrate some of the basic features of python syntax:
  - I can add comments using the # character
  - = is used to assign values to variables
  - To do operations between numbers and variables I can use the usual operators +, -, *, /

```
x = 34 - 23 # commentare il codice
y = "Hello"
z = 3.45
```

Python Basics

- Let's see a simple code that allows us to illustrate some of the basic features of python syntax:
    - == is the operator that server compare values
    - The logical operators are instead: and, or, not
    - The + operator can also be used to concatenate strings



```
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"
```

Python Basics

- Let's see a simple code that allows us to illustrate some of the basic features of python syntax:
  - print is the basic command used to print on the screen
  - **Variables do not need to be explicitly declared the first time that I assign a value the variable is created and given a type**

```
print("Valore di x: ", x)
print("Valore di y: ", y)

print(z)
```

Python Basics

```
x = 34 - 23 # commentare il codice
y = "Hello"
z = 3.45

if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"

print("Valore di x: ", x)
print("Valore di y: ", y)

print(z)
```

```
Valore di x:  12
Valore di y:  Hello World
3.45
```

Python Basics

- There is no end-of-line character, if a line of code has to be broken on several lines, use \
- **Python 2** By default the numbers are integers so z = 5/2 will give as a result 2

```
└ $ cat oltrehw2.py
z = 5 / 2
print(z)

z = 5.0 / 2.0
print(z)
|redo@buchner /home/redo/Lezio
└ $ python2 oltrehw2.py
2
2.5
|redo@buchner /home/redo/Lezio
└ $ python3 oltrehw2.py
2.5
2.5
```

IF … THEN … ELSE

If … then … else

INSERT NUMBER I

IF I LOWER THEN 0

PRINT “il numero è minore di zero”

ELSE IF I EQUAL TO 0

PRINT “Il numero è uguale a zero”

ELSE

PRINT “Il numero è maggiore di zero”

ENDIF

```
                  [ Start ]
                      |
                      v
              [ Input Number I ]
                      |
                      v
                < Is I < 0? >
                  /       \
               Yes         No
               /             \
              v               v
[Print "il numero        < Is I == 0? >
 è minore di zero"]        /        \
        |               Yes          No
        |               /              \
        |              v                v
        |     [Print "Il numero   [Print "Il numero
        |      è uguale a zero"]   è maggiore di zero"]
        |              |                |
         \             |               /
          \            |              /
           \           |             /
            v          v            v
                  [ End ]
```

If ... then ... else

- To identify blocks of code in python, empty spaces are used, not for example the {} as in C / C ++

```
└─ $ cat oltrehw4.py
si = input ("inserisci un numero: " )
i = float(si)

if i < 0 :
    print("numero inferiore a zero")
elif i == 0:
    print("inseiro uguale a zero")
else:
    print("numero maggiore di zero")
```

```
inserisci un numero: -31
numero inferiore a zero
inserisci un numero: 3.0
numero maggiore di zero
```

If … then … else

- You can use eval()

```
si = eval(input ("inserisci un numero: " )) # The expression argument
                                            # is parsed and evaluated as a
                                            # Python expression (technically
                                            # speaking, a condition list)
                                            # using the globals and locals
                                            # dictionaries as global and local
                                            # namespace.


if si < 0 :
    print("numero inferiore a zero")
elif si == 0:
    print("inseiro uguale a zero")
else:
    print("numero maggiore di zero")
```

```
└─ $ python3 oltrehw4.py
inserisci un numero: 10
numero maggiore di zero
inserisci un numero: print("si ", si)
si  10
Traceback (most recent call last):
  File "oltrehw4.py", line 20, in <module>
    if si < 0 :
TypeError: '<' not supported between instances of 'NoneType' and 'int'
```

LOOPS

Loops

SET N TO 0

SET n TO 0

Repeat the following:

a. If n >= 10, terminate the repetition, otherwise.
b. Increment N by n
c. PRINT n

PRINT N

```
                  [ Start ]
                      |
                      v
                [ SET N = 0 ]
                [ SET n = 0 ]
                      |
                      v
+--------------> < Is n >= 10? >
|                    /         \
|                  No           Yes
|                  /              \
|                 v                v
|          [ N = N + n ]      [ PRINT N ]
|                 |                |
|                 v                v
|            [ PRINT n ]        [ End ]
|                 |
|                 v
|           [ n = n + 1 ]
|                 |
+-----------------+
```

Loops

- To identify blocks of code in python, empty spaces are used, not for example the {} as in C / C ++
- **Python is case sensitive**

```
└ $ less oltrehw3.py
N = 0

for n in range(0,10):
  N = N + n
  print(n)

print("Valore finale: ", N)
```

Loops

- To identify blocks ... d, not for example the {} as in C / C...
- **Python is case ...**

```
N = 0

for n in range(0,10):
  N = N + n
  print(n)

print("Valore finale: ", N)
```

```
0
1
2
3
4
5
6
7
8
9
Valore finale:  45
```



```
print("Valore finale: ", N)
```

EXAMPLE

Example of a numerical procedure

It is possible to find an algorithm to solve almost any problems, but not all. For example, calculate the solutions of a second-order equation:

```
INPUT  A, B, C
COMPUTE D = (B*B)-(4 * A * C)
IF D >= 0.0
  SOL1 = (-1,0 * B + SQRT(D)) / (2,0 * A)
  SOL2 = (-1,0 * B - SQRT(D)) / (2,0 * A)
  PRINT SOL1 AND SOL2
ELSE
  PRINT “non ci sono soluzioni reali”
```

```
          [ INPUT A, B, C ]
                  |
                  v
    [ COMPUTE D = (B*B) - (4*A*C) ]
                  |
                  v
           < Is D >= 0.0? >
             /          \
          Yes            No
           /              \
          v                v
[ SOL1 = (-1.0*B + SQRT(D))  [ PRINT "non ci sono
     / (2.0*A) ]              soluzioni reali" ]
          |                    |
          v                    |
[ SOL2 = (-1.0*B - SQRT(D))    |
     / (2.0*A) ]               |
          |                    |
          v                    |
[ PRINT SOL1 AND SOL2 ]        |
          |                    |
           \                  /
            \                /
             v              v
               [ End ]
```

Example of a numerical procedure

```
└ $ cat solv.py
import math

a = float(input("insert a:"))
b = float(input("insert b:"))
c = float(input("insert c:"))

print("a = ", a, " b = ", b, " c = ", c)

delta = math.pow(b, 2.0) - (4.0 * a * c)

if (delta >= 0):
  tn = math.sqrt(delta)
  sol1 = ((-1.0 * b) + tn) / (2.0 * a)
  sol2 = ((-1.0 * b) - tn) / (2.0 * a)
  print(sol1, sol2)
else:
  print("No real solutions")
```

Example of a numerical procedure

```
└─ $ cat solv.py
import math

a = float(input("insert a:"))
b = float(input("insert b:"))
c =

pri

del

if
  t
  sol1 = ((-1.0 * b) + tn) / (2.0 * a)
  sol2 = ((-1.0 * b) - tn) / (2.0 * a)
  print(sol1, sol2)
else:
  print("No real solutions")
```

```
    c = float(c1)
```

Console

```
<terminated> main.py [/usr/bin/python]
insert a:1
insert b:15
insert c:2.5
a =  1.0  b =  15.0  c =  2.5
-0.168560850692 -14.8314391493
```

EXERCISE 1

EXERCISE 1

Write a program in python that reads 10 numbers, after it calculates the average value and prints the result

```
[redo@banquo esercizipython (master)]$ python ex1.py
inserisci il numero 1 45
inserisci il numero 2 67
inserisci il numero 3 84
inserisci il numero 4 2
inserisci il numero 5 4
inserisci il numero 6 6
inserisci il numero 7 7
inserisci il numero 8 8
inserisci il numero 9 9.0
inserisci il numero 10 13.0
la somma:  245.0
valore medio:  24.5
[redo@banquo esercizipython (master)]$
```

BREAK

Break

The usually syntactically nested break in a for loop or while, ends the nearest loop, skipping the optional else clause if the loop has one.

```
└ $ cat testbreaks.py
for i in range(0,10):
    print("i: ", i)
    if i > 5:
        break;
```

```
i:  0
i:  1
i:  2
i:  3
i:  4
i:  5
i:  6
```

Break and nested loops

```
└─ $ cat testbreaks1.py
for i in range(10):
    print("loop 1: ", i)
    for j in range(10):
        print("     loop 2: ", j)
        if (j > 5):
            break;
```

```
loop 1:  0
    loop 2:  0
    loop 2:  1
    loop 2:  2
    loop 2:  3
    loop 2:  4
    loop 2:  5
    loop 2:  6
loop 1:  1
    loop 2:  0
    loop 2:  1
    loop 2:  2
    loop 2:  3
    loop 2:  4
    loop 2:  5
    loop 2:  6
loop 1:  2
    loop 2:  0
    loop 2:  1
```

Break and nested loops

```
└ $ cat testbreaks2.py
for i in range(10):
    print("1) loop 1: ", i)
    for j in range(10):
        print("    loop 2: ", j)
        if (j > 5):
            break;
    print("2) loop 1: ", i)
```

```
1) loop 1:  0
    loop 2:  0
    loop 2:  1
    loop 2:  2
    loop 2:  3
    loop 2:  4
    loop 2:  5
    loop 2:  6
2) loop 1:  0
1) loop 1:  1
    loop 2:  0
    loop 2:  1
    loop 2:  2
    loop 2:  3
    loop 2:  4
    loop 2:  5
    loop 2:  6
2) loop 1:  1
1) loop 1:  2
```

RANDOM NUMBER

Random Number

- Pseudo Random vs Random
  - while true; do cat /dev/random | od -vAn -N2 -tu; done

```
└ $ while true; do cat /dev/random | od -vAn -N2 -tu; done
 11063
  8852
 35128
 53522
 63860
 53034
 37027
 43798
 16198
[SYSTEM] 0:bash- 1:[tmux]* 2:bash
```

Random Number

Standard libraries, like Python's random module, use PRNGs (specifically, the Mersenne Twister algorithm).

The Illusion of Randomness

These numbers are **not truly random**. They are generated by a complex, deterministic mathematical formula. If you know the starting point (the *Seed*), you can perfectly predict every "random" number that will ever be generated.

Random Number

Standard libraries, like Python's random module, use PRNGs (specifically, the Mersenne Twister algorithm).

```
import random

# Set the "Seed" (The starting state of the math formula)
random.seed(42)

print(random.randint(1, 100)) # Always prints 82
print(random.randint(1, 100)) # Always prints 15
print(random.randint(1, 100)) # Always prints 4
```

Random Number

```
import random

for i in range(100):
  print(random.randint(0,10))
```

```
9
3
3
4
2
4
0
5
4
10
10
2
3
10
```

EXERCISE 2

Exercise 2

Write a program that generates a random number R between 0 and 20 and asks to the user to guess the number with a maximum of 10 attempts. Each time the program will simply writes if the number inserted is greater or less than R. Clearly if the inserted number is equal to the generated R random number the program will exit

```
inserisci numero: 10
il numero inserito e' troppo piccolo
inserisci numero: 18
il numero inserito e' troppo grande
inserisci numero: 15
il numero inserito e' troppo grande
inserisci numero: 12
bravo indovinato
```

Pseudocode

```
GENERATE A RANDOM NUMBER rnd

Repeat the following:
    INPUT b
    IF b IS EQUAL TO rnd
        PRINT “well done”
        BREAK
    ELSE IF b < rnd
        PRINT “inserted number is too small”
    ELSE
        PRINT “inserted number is too big”
    ENDIF
```

```
                      [ Start ]
                          |
                          v
+-----------> [ INPUT b ]
|                         |
|                         v
|                 < Is b == rnd? >
|                   /             \
|                 No               Yes
|                /                   \
|               v                     v
|     < Is b < rnd? >       [ PRINT "well done" ]
|      /             \                 |
| Yes                 No               |
| /                    \               |
|v                      v              |
|[ PRINT "inserted    [ PRINT "inserted
|  number is too        number is too
|     small" ]             big" ]
|         |                  |
+---------+------------------+
                               |
                               v
                            [ End ]
```

Contents

- Brief introduction to programming languages and computer complexity
  - Python
  - Python data structures and more

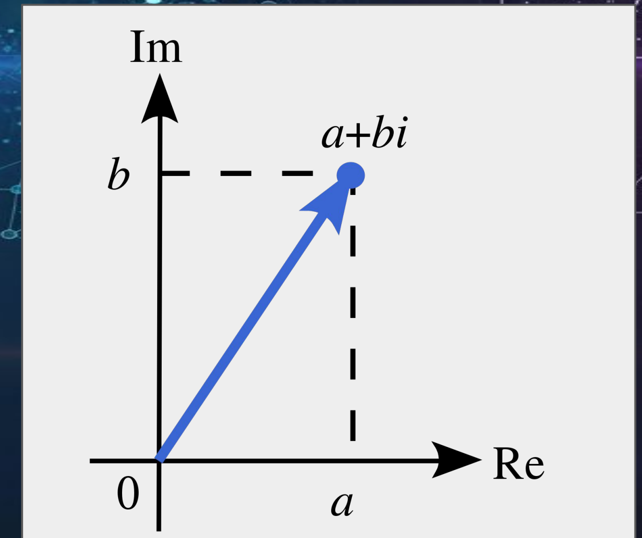
Data structures

- Data structures allow to organize data in order to make their use and their handling more efficient
- We will see in particular strings, lists, tuples, dictionaries and the sets.
- We will only see the minimum bases useful to carry out basic exercises

COMPLEX NUMBERS

Complex numbers

A complex number is a number that can be expressed in the form **a + ib**, where **a** and **b** are real numbers, and **i** is a solution of the equation $x^2 = -1$. Because no real number satisfies this equation, **i is called an imaginary number**.



Complex numbers

- Complex numbers are intrinsically defined in python

```
a = 2 + 3j
b = 4 - 1j

print(a)
print(a.real, " ", a.imag)
print(b)
print(b.real, " ", b.imag)
```

```
(2+3j)
2.0   3.0
(4-1j)
4.0   -1.0
```

Complex numbers

- Complex numbers and the type function type() function is mostly used for debugging purposes. Two different types of arguments can be passed to type() function, single and three argument. If single argument type(obj) is passed, it returns the type of given object.

```
c = a * b
print(type(c), " valore ", c)

if type(c) == complex:
    print("c e\' un numero complesso")
```

```
<class 'complex'>  valore  (11+10j)
c e' un numero complesso
```

STRINGS

Strings

- A string is a sequence of characters, in python there are several methods / operations useful for the manipulation of strings, **a string in general can be seen as an array of characters**
- Operations between strings

```
str1 = "hello"
str2 = "world"

print(str1 + " " + str2)
print(3*(str1+" "))
```

```
hello world
hello hello hello
```

Strings

- Extract sub-strings, strings are arrays of characters

Strings : delimiter

- : is the delimiter of the slice syntax to 'slice out' sub-parts in sequences , [start:end]
- [1:5] is equivalent to "from 1 to 5" (5 not included)
- [1:] is equivalent to "1 to end"
- [:-2] from “begin to -2” -2 not included

H	e	l	l	o
0	1	2	3	4
-5	-4	-3	-2	-1

```
str1 = "Hello"
print(str1[0])
print(str1[0:3])
print(str1[-2:])
print(str1[-1])
print(str1[:-2])
```

```
H
Hel
lo
o
Hel
```

Strings

Extract sub-strings, strings are arrays of characters

```
teststr = "Hello, World!"

for i in range(0,len(teststr)):
    print(teststr[i])
```

```
H
e
l
l
o
,

W
o
r
l
d
!
```

Strings

String's class in python has several methods

- **find()** It determines if string str occurs in string, or in a substring of string if starting index beg and ending index end are given.

```
str = "Hello, World!"
index = str.find("ll")
if index >= 0:
  print("a ", index, " trovato ", str[index:])
```

```
a  2  trovato  llo, World!
```

Strings

- **split()** method returns a list of strings after breaking the given string by the specified separator.

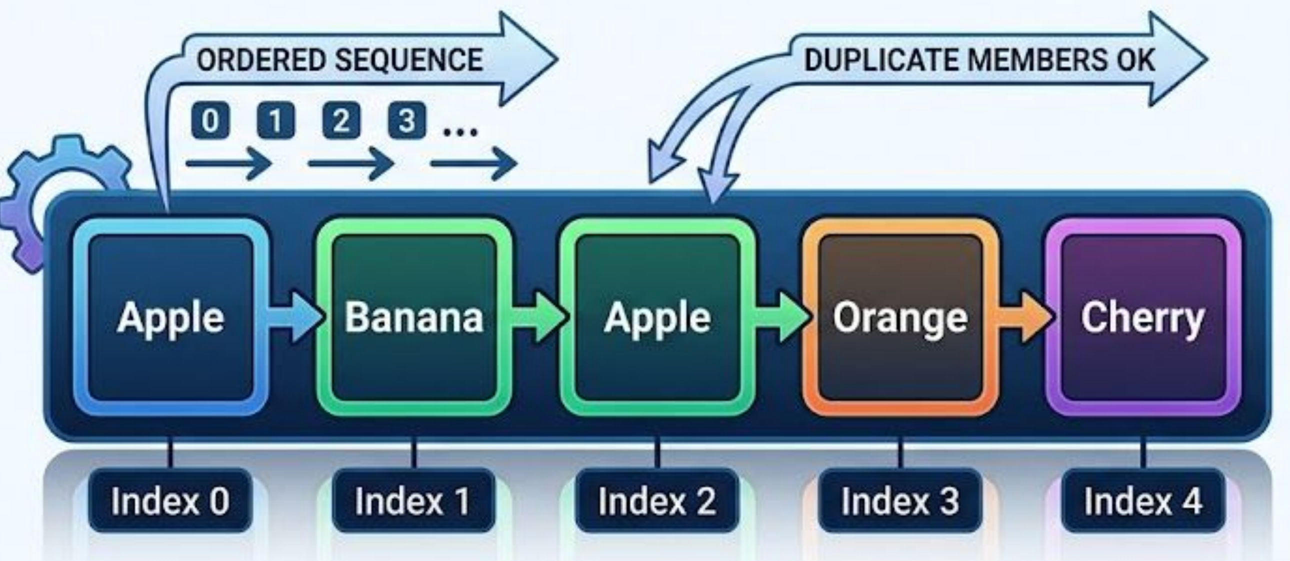
```
str = "Hello, World!"
res = str.split(" ")
idx = 0
for r in res:
    idx += 1
    print(idx , " - ", r)
```

```
1  -  Hello,
2  -  World!
```

LISTS

THE LIST DATA STRUCTURE

An **ordered** and **changeable** collection. Allows duplicate members.



Lists

- Lists are ordered sequences of objects, many basic operations are in common with strings

```
a = [1, "pippo", 4.5, "pluto"]
print(a[0])
for i in range(0,len(a)):
    print(a[i])

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

```
1
1
pippo
4.5
pluto
1
pippo
4.5
pluto
```

Lists

- Lists are ordered sequences of objects, many basic operations are in common with strings

```
a = [1, 3.5, -6.0, 5]
print(a)
a[1] = "pluto"
print(a)
```

```
[1, 3.5, -6.0, 5]
[1, 'pluto', -6.0, 5]
```

Lists

Lists are ordered sequences of objects, many basic operations are in common with strings, but remember:

A string does not support item assignment

```
>>> str = "Hello"
>>> str[0] = "v"
Traceback (most recent
  File "<stdin>", line
TypeError: 'str' objec
>>> str = "pluto"
```

Lists

Lists have many methods we will see only the most important

- **list.append(elem)** -- adds a single element to the end of the list.
- **list.insert(index, elem)** -- inserts the element at the given index, shifting elements to the right.
- **list.extend(list2)** adds the elements in list2 to the end of the list.

Lists

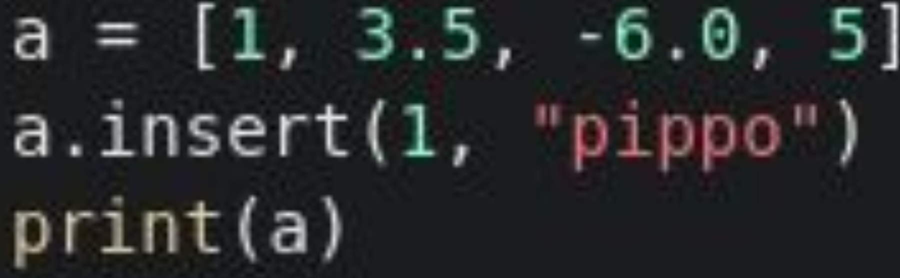
Lists adding elements

```
a = [1, 3.5, -6.0, 5]
a.append(46)
a.append("pluto")
print(a)
```

```
[1, 3.5, -6.0, 5, 46, 'pluto']
```

Lists

Lists adding elements



```
a = [1, 3.5, -6.0, 5]
a.insert(1, "pippo")
print(a)
```

```
[1, 'pippo', 3.5, -6.0, 5]
```

Lists

Lists adding elements

```
a = [1, 3.5, -6.0, 5]
b = ["second", "list"]
a.extend(b)
print(a)
```

```
[1, 3.5, -6.0, 5, 'second', 'list']
```

Lists

Remove elements from a list

1. **remove** removes the first matching value, not a specific index

```
a = [1, 2, 3, 4, 2, 5]
a.remove(2)
print(a)
a.remove(2)
print(a)
```

```
[1, 3, 4, 2, 5]
[1, 3, 4, 5]
```

Lists

Remove elements from a list

- **del** removes the item at a specific index

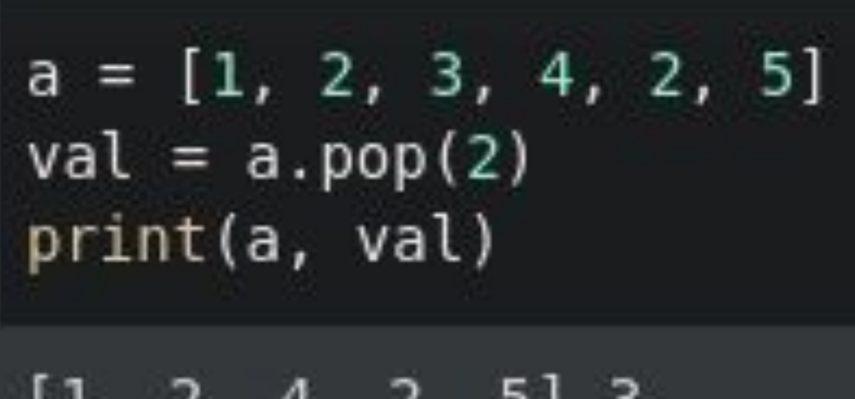
```
a = [1, 2, 3, 4, 2, 5]
del a[2]
print(a)
del a[2]
print(a)
```

```
[1, 2, 4, 2, 5]
[1, 2, 2, 5]
```

Lists

Remove elements from a list

1. **pop** removes the item at a specific index and returns it



Lists

List has a **sort** method

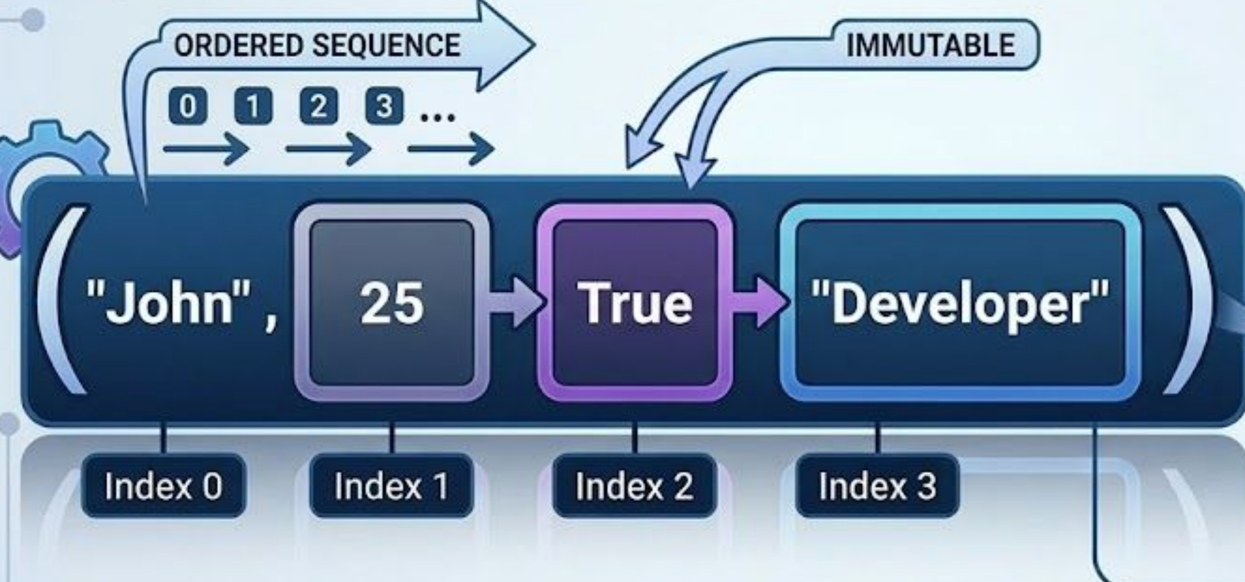
```
a = [1, 4, 3, 5, 9, 34, 7]
a.sort()
print(a)
c = ["a", "pluto", "paperino", "b"]
c.sort()
print(c)
```

```
[1, 3, 4, 5, 7, 9, 34]
['a', 'b', 'paperino', 'pluto']
```

TUPLES

THE TUPLE DATA STRUCTURE

An **ordered**, **immutable** collection of elements.



TUPLE PROPERTIES

1. IMMUTABLE (UNCHANGEABLE)

Elements cannot be added, removed, or modified after creation.

Add:

Remove:

Update:

2. HETEROGENEOUS ELEMENTS

"John" - 25 - True

string integer boolean

Can store different data types together.

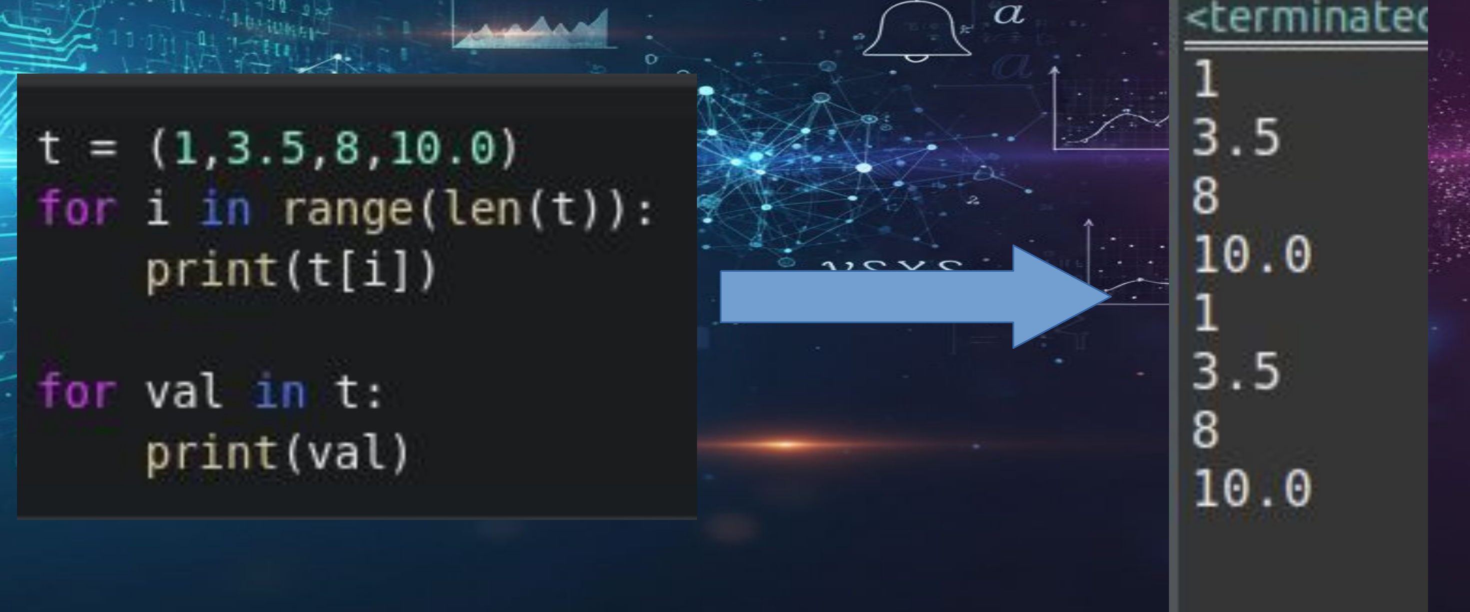
USE CASES

(x, y) or (ID, Name, Date)

Representing fixed records or coordinates.

Tuples

Python tuples are very similar to lists but their manipulation is faster since they are "**immutable**"



Tuples

But I can not modify a value

```
# ma posso modificare un valore ?
t[1] = 0
```

```
---------------------------------------------------------------
TypeError                                 Trace
<ipython-input-30-342d053316bd> in <module>()
      1 # ma posso modificare un valore ?
----> 2 t[1] = 0

TypeError: 'tuple' object does not support item
```

SEARCH STACK OVERFLOW

But...

But a tuples element can be a mutable one so...

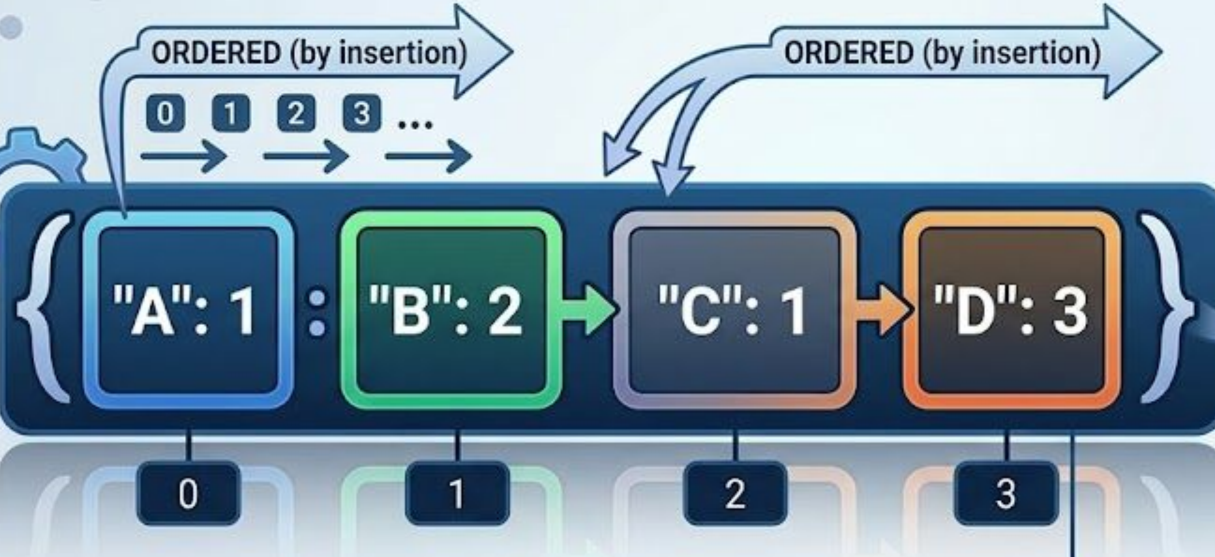
```
t = (1, 4, 5, 6 , "se", 4, [7, 9, 0])
print(t)
t[6].append("last one")
print(t)
t[6][0] = "primo"
print(t)
```

```
(1, 4, 5, 6, 'se', 4, [7, 9, 0])
(1, 4, 5, 6, 'se', 4, [7, 9, 0, 'last one'])
(1, 4, 5, 6, 'se', 4, ['primo', 9, 0, 'last one'])
```

DICTIONARY

THE DICTIONARY DATA STRUCTURE

An **insertion-ordered**, **changeable** collection of key-value pairs. Unique



DICTIONARY PROPERTIES

1. MUTABLE (CHANGEABLE)

Pairs can be added, removed, or values modified after creation.

Update Value Add Pair Remove Pair

2. UNIQUE KEYS, DUPLICATES OK

Keys must be unique and hashable. Values can be duplicates.

USE CASES

{'speed': 100, 'type': 'car'}
{'John': '555-1212', 'Mary': '555-3434'}

Fast lookups, mappings, structured data records.

Dictionary

A dictionary is a **sequence of elements**, **each element is a pair key : value.** The keys are unique and dictionaries are **created using {**

```
d = {"k1" : 1, "k2" : 2, 4 : "val3"}
print(d[4], d["k1"])
d["k1"] = 1.5
d["quattro"] = 4
print(d)
```

```
val3 1
{'k1': 1.5, 'k2': 2, 4: 'val3', 'quattro': 4}
```

Dictionary

We can check if a key is present or not easily

```
diz = {'k1': 1, 'k2': 2, 4: 'val3', 'quattro': 4}
if "quattro" in diz:
  print("yes and value is ", diz["quattro"])
if not 5 in  diz:
  print("key it is not present")
```

```
yes and value is  4
key it is not present
```

Dictionary

We can check if a value is present or not easily

```
diz = {'k1': 1, 'k2': 2, 4: 'val3', 'quattro': 4}

if "val3" in diz.values():
  print("value value3 is present")
```

```
value value3 is present
```

Dictionary

dict.items() returns of the dictionary a list of tuples

```
diz = {'k1': 1, 'k2': 2, 4: 'val3', 'quattro': 4}
for x in diz.items():
  print(x)
```

```
('k1', 1)
('k2', 2)
(4, 'val3')
('quattro', 4)
```

Dizionari

```python
diz = {'k1': 1, 'k2': 2, 4: 'val3', 'quattro': 4}
for x in diz.items():
  print(x)
for k in diz.keys():
  print(k)
for  v in diz.values():
  print(v)
```

```
('k1', 1)
('k2', 2)
(4, 'val3')
('quattro', 4)
k1
k2
4
quattro
1
2
val3
4
```

Dizionari

clear(): removes all items

pop(): removes and returns element having given key

del statement removes the given item from the dictionary. If given key is not present in dictionary then it will throw KeyError.

```
diz = {'k1': 1, 'k2': 2, 4: 'val3', 'quattro': 4}
del diz["k1"]
print(diz)
v = diz.pop(4)
print(diz, v)
diz.clear()
print(diz)
```

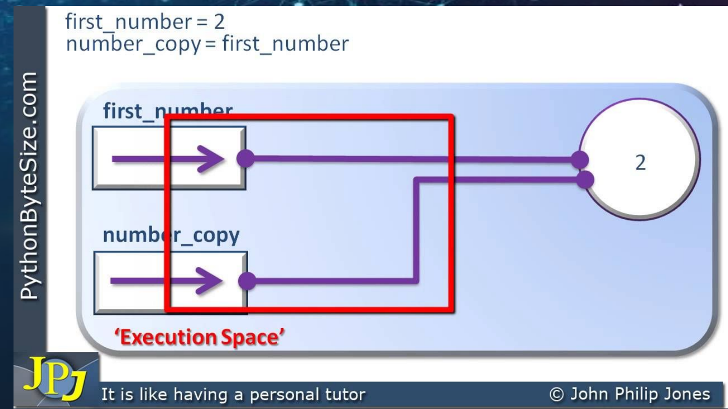
```
{'k2': 2, 4: 'val3', 'quattro': 4}
{'k2': 2, 'quattro': 4} val3
{}
```

PYTHON SOME DETAILS ABOUT MEMORY MANAGEMENT

Reference

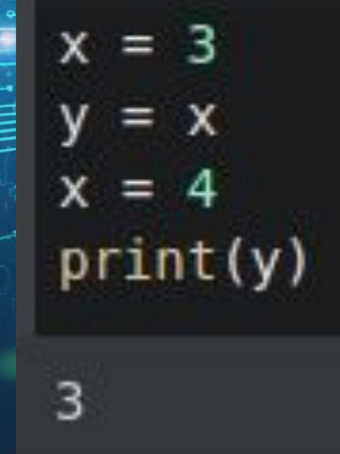
When I declare a variable I am asking for a certain amount of memory

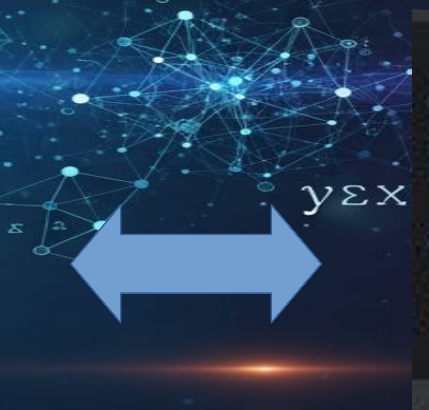
- In python, the assignment operation manipulates the references, so **x = y** does not create a copy of the data contained in **y**, but simply creates a reference to **y**, that is **x** will point to **y**

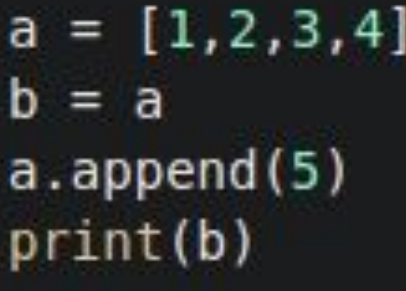


Reference

When I write x = 4 we allocate the memory space necessary to contain the integer 4 and then we "stored the address of the" (created the reference to) the memory location in x



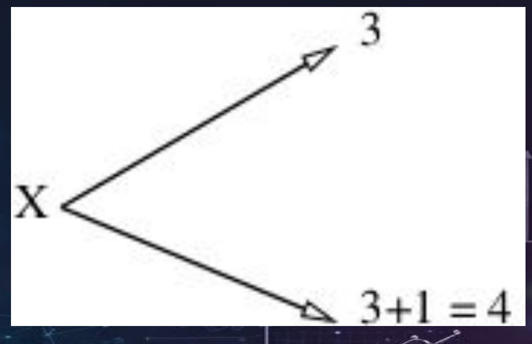




Reference

```
x = 3
x = x + 1
print(x)
```

```
4
```



What really happens when increment x?

- The interpreter retrieves the value contained in the memory address to which x refers
- The result of the 3 + 1 operation is calculated and the result is stored in a new memory location
- You change the reference in x, x will now refer to the new address in memory where the value 4 is stored
- Python has a garbage collector that eliminates free all the memory allocated when there are no more names referring to the memory areas in question

EXECISE MATRIX MULTIPLICATION

Matrix

I can use a list of lists to store a matrix in a simple way

```
import random
import math

A = [[0.0, 0.0, 0.0], \
     [0.0, 0.0, 0.0], \
     [0.0, 0.0, 0.0]]

for i in range(len(A)):
  for j in range(len(A[0])):
    A[i][j] = random.uniform(0.0, 1.0)

print("Matrix A")
print(A)
```

```
Matrix A
[[0.109058751134825, 0.24607185195192582, 0
```

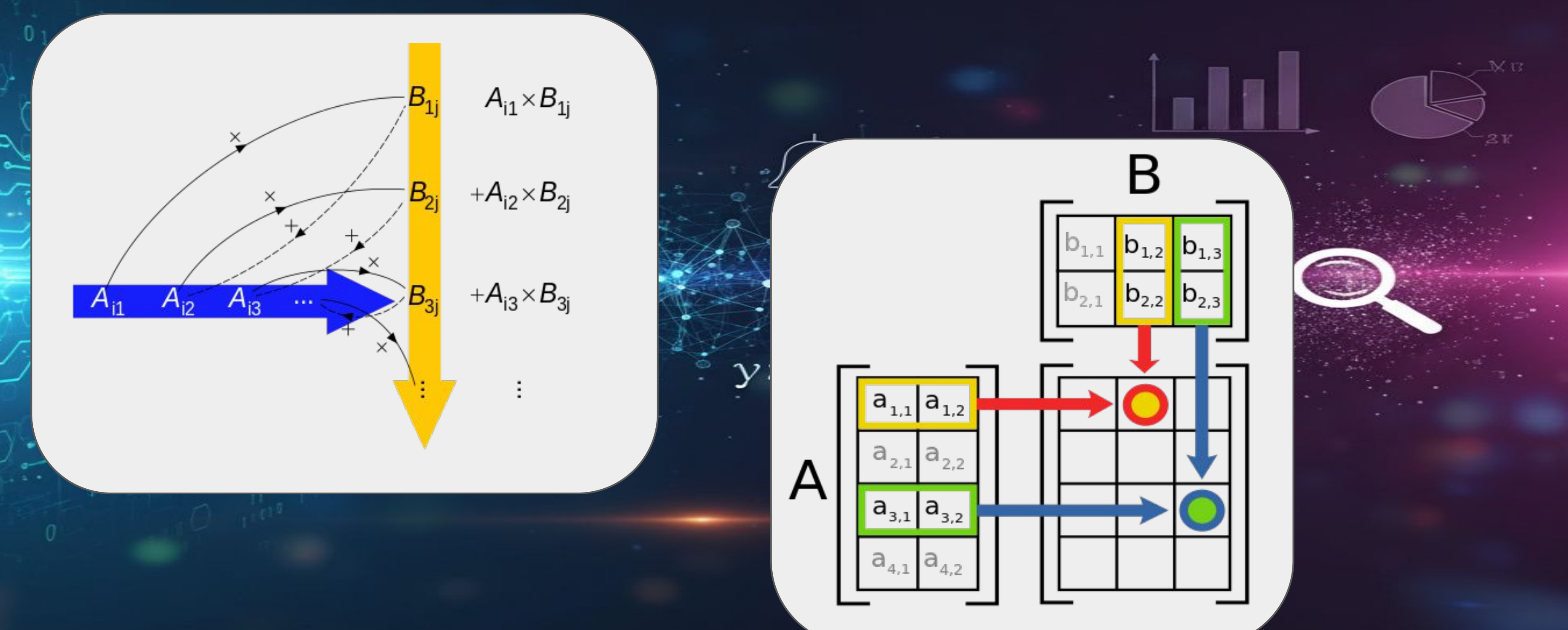
Matrix multiplication

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \,.$$

The product is defined only for matrices with compatible dimensions

Moltiplicazione matrice matrice



Matrix Multiplication

Write a program that given two 3x3 arrays filled with random numbers computes and prints the result of the matrix multiplication

```
[redo@banquo mtxmtx (master)]$ python mtx.py
Matrix A
[0.19246968384248186, 0.9696947765114629, 0.8237325914685499]
[0.177986003994452, 0.9755353119130369, 0.8217085413339825]
[0.41279486028220536, 0.05779395844699264, 0.40208982438481 4]
Matrix B
[0.7196544280415835, 0.8235257700392403, 0.934926373722345 8]
[0.27393524261728885, 0.7093719818717363, 0.440755172029067]
[0.12608819669592142, 0.939121875851728, 0.4639718068159948]
Matrix C
[0.5080081911273185, 1.6199633465203456, 0.9895316704002202]
[0.4989296664411048 7, 1.610277945334311 2, 0.977625640109390 6]
[0.363600231970347 2, 0.758555970163058 2, 0.597964130234557 9]
```

Matrix multiplication simple algorithm

Input: matrices A and B

Let C be a new matrix of the appropriate size

For i from 0 to n-1:

For j from 0 to p-1:

Let sum = 0

For k from 0 to m-1:

Set sum = sum + $A_{ik} \times B_{kj}$

Set C_{ij} = sum

Return C

Matrix multiplication simple algorithm

```python
import random

A =  [[0.0, 0.0, 0.0],
      [0.0, 0.0, 0.0],
      [0.0, 0.0, 0.0]]

B =  [[0.0, 0.0, 0.0],
      [0.0, 0.0, 0.0],
      [0.0, 0.0, 0.0]]

C =  [[0.0, 0.0, 0.0],
      [0.0, 0.0, 0.0],
      [0.0, 0.0, 0.0]]
```

```python
for i in range(len(A)):
  for j in range(len(A[i])):
    A[i][j] = random.uniform(0.0, 1.0)

for i in range(len(B)):
  for j in range(len(B[i])):
    B[i][j] = random.uniform(0.0, 1.0)

dimension = len(A)
assert(len(B) == dimension)
assert(len(C) == dimension)
for i in range(dimension):
  assert(len(A[i]) == dimension)
  assert(len(B[i]) == dimension)
  assert(len(C[i]) == dimension)
```

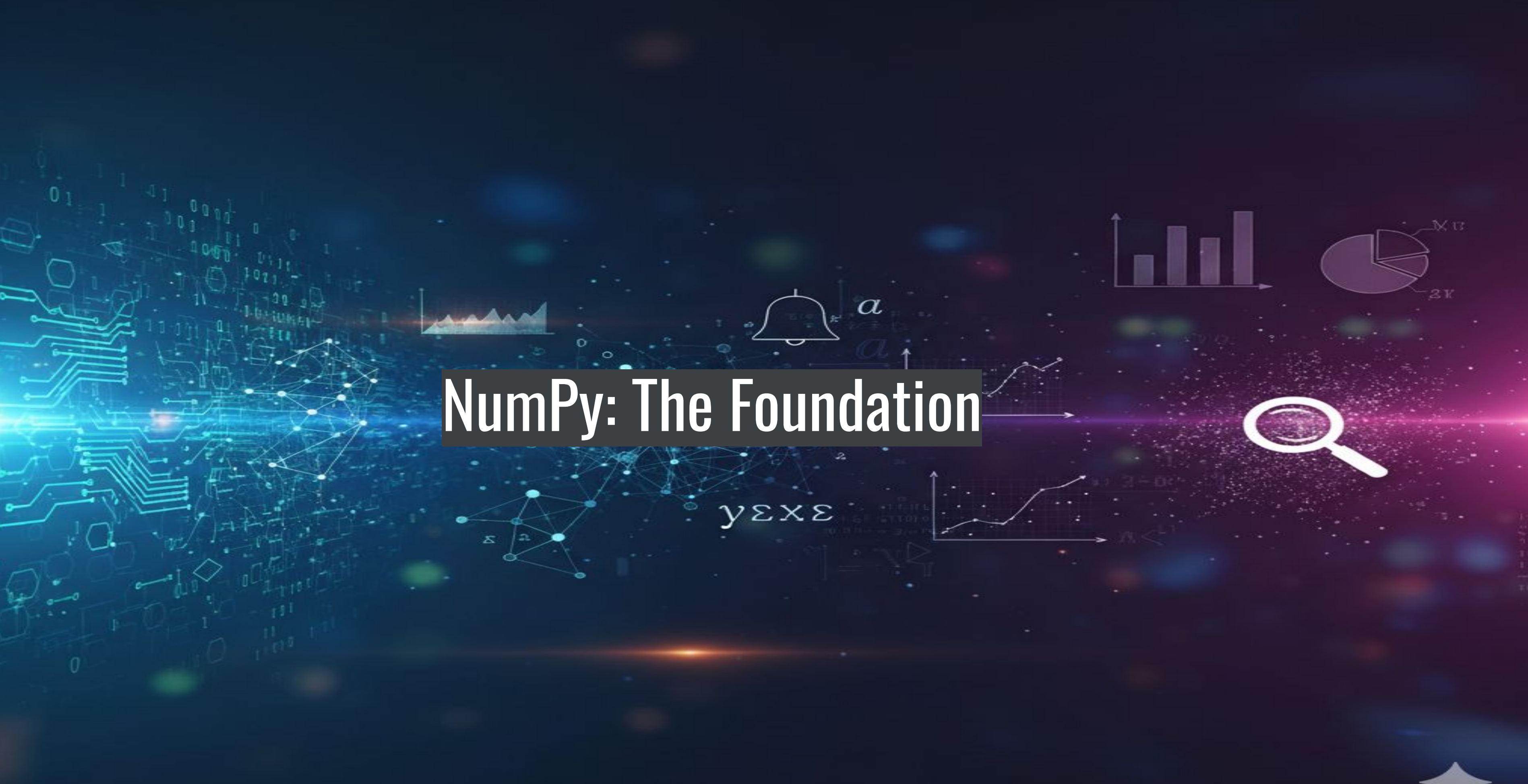
Matrix multiplication simple algorithm

```
for i in range(len(A)):
  for j in range(len(B[0])):
    for k in range(len(B)):
      C[i][j] = C[i][j] + A[i][k]*B[k][j]

print("Matrix A")
for i in range(len(A)):
  print(A[i])
```

Matrix multiplication simple algorithm

```
Matrix A
[0.18146819893329935, 0.7950506482410533, 0.5185755611584035]
[0.2132423302815008, 0.3648003922144545, 0.679296272743230б]
[0.9684603995939413, 0.35700036600823515, 0.9210188380704635]
Matrix B
[0.010086455499074276, 0.9950179801549787, 0.39983594023928604]
[0.743445981538309 7, 0.135524675757991, 0.2492650920073761 2]
[0.722166889339538, 0.010856051889053 92, 0.749332827998264 8]
Matrix C
[0.967405680356546 7, 0.293942785279353 5, 0.6593215727019694]
[0.7639255211572 6, 0.2689938832164323 6, 0.685212948059948 5]
[0.940308129552110 5, 1.022016497809686 2, 1.166362654106884 1]
```

NumPy: The Foundation



What is NumPy?

NumPy (Numerical Python) is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

- **Contiguous Memory:** Unlike Python lists, NumPy's ndarray stores elements in contiguous memory blocks.
- **Blazing Fast:** Written partially in C, NumPy executes numerical operations up to 50x faster than standard Python lists.
- **Homogeneous Data:** All elements in a NumPy array must be of the exact same data type (e.g., all floats or all integers).

Creating Arrays

Standard Initialization

The most common way to create a NumPy array is by converting a standard Python list using np.array().

```python
import numpy as np

# From a standard Python list
my_list = [1, 2, 3, 4, 5]
arr = np.array(my_list)

print(arr)
# Output: [1 2 3 4 5]

print(type(arr))
# Output:
```

Built-in Generators

NumPy provides extremely efficient built-in functions to generate arrays filled with specific patterns.

```python
# Array of exactly ten 0s
zeros = np.zeros(10)

# Array from 0 to 9 (like Python range)
seq = np.arange(10)

# 5 evenly spaced numbers between 0 and 1
space = np.linspace(0, 1, 5)
# [0. 0.25 0.5 0.75 1. ]
```

Array Dimensions & Shapes

NumPy's true power lies in its ability to handle N-dimensional data structures effortlessly.

1D Array (Vector)

A single line of values. Useful for simple time series or sequential data.

`shape = (4,)`

2D Array (Matrix)

Rows and columns. The foundation of tabular data, images (grayscale), and linear algebra.

`shape = (3, 4)`

3D Array (Tensor)

A "stack" of matrices. Used extensively for RGB images, video frames, and deep learning.

`shape = (2, 3, 4)`

Use arr.ndim to get the number of dimensions, and arr.shape to get a tuple representing the size of each dimension.

The Power of Vectorization

No More "For Loops"

In standard Python, adding two lists together concatenates them. If you want to add their mathematical elements together, you must write a slow for loop.

NumPy allows **Vectorization**: performing mathematical operations on entire arrays simultaneously at C-level speeds.

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Element-wise addition
print(a + b)        # [5 7 9]

# Scalar multiplication
print(a * 10)       # [10 20 30]
```

Indexing and Slicing

Basic Slicing (1D & 2D)

NumPy slicing syntax follows Python's standard [start:stop:step] mechanism, but scales to multiple dimensions.

```
arr = np.arange(10)
# [0 1 2 3 4 5 6 7 8 9]

# Grab elements from index 2 to 5
print(arr[2:5])  # [2 3 4]

# 2D Array Slicing [row, col]
matrix = np.array([[1,2,3], [4,5,6]])
print(matrix[0, 1:3]) # [2 3]
```

Boolean Masking

A powerful feature where you filter an array using conditional logic, returning only the elements that evaluate to True.

```
arr = np.array([15, 4, 8, 22, 7])

# Create a mask of booleans
mask = arr > 10
print(mask)
# [True False False True False]

# Apply the mask
print(arr[mask]) # [15 22]

# Shorthand notation
print(arr[arr < 10]) # [4 8 7]
```

Unmatched Performance

Why is it so fast?

When processing massive datasets, standard Python lists fail because they store memory addresses pointing to scattered objects in RAM.

- **C-Backend:** NumPy core operations are written in highly optimized C code.
- **Locality of Reference:** Data is stored in contiguous, unbroken blocks of memory.
- **SIMD Instructions:** Modern CPUs process vectorized NumPy arrays in parallel hardware tracks.

From Built-in I/O to Pandas

Python Built-in File I/O

Before jumping into heavy libraries, Python has native, built-in functions for handling raw text files. The primary tool is the `open()` function.

- **Syntax:** `open('filename.txt', 'mode')`
- **Memory Management:** Once a file is opened and processed, it must be closed using `file.close()` to free up system resources.
- **Raw Data:** Standard I/O processes everything as raw strings or bytes.

Mode	Description
'r' (Read)	Default mode. Opens for reading. Fails if the file does not exist.
'w' (Write)	Opens for writing. **Overwrites** existing content or creates a new file.
'a' (Append)	Opens for writing. Appends new data to the **end** of the file.
'r+' (Read/Write)	Opens a file for both reading and updating without truncating.

Reading & Writing Files

The safest and most Pythonic way to handle files is using the `with` statement (Context Manager), which automatically closes the file for you.

Writing Data

Writing strings directly to a text file.

```
# The 'with' block manages the file lifecycle
with open('output.txt', 'w') as f:
    f.write("Hello, Python I/O!\n")
    f.write("This is the second line.")

# The file is automatically closed here!
```

Reading Data

Reading raw text line by line.

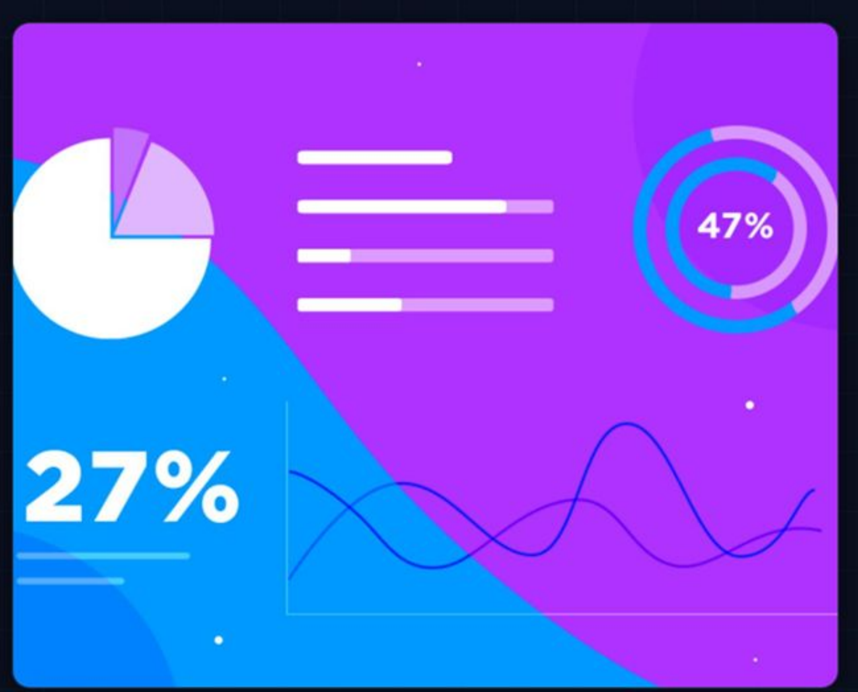
```
with open('output.txt', 'r') as f:
    # Option A: Read the entire file
    content = f.read()

    # Option B: Loop through line by line
    for line in f:
        print(line.strip())
```

The Limitation of Standard I/O

Standard I/O is great for plain text, but terrible for tabular data (like CSVs). If you use native Python to read a spreadsheet, you have to manually parse everything.

```
# The Native Python Way (Tedious)
with open('data.csv', 'r') as f:
    for line in f:
        row = line.strip().split(',')
        age = int(row[1]) # Manual casting
```

When data becomes structured, complex, and massive, we must graduate from standard I/O to a dedicated data manipulation library: Pandas.



What is pandas?

`pandas` is a fast, powerful, and flexible open-source data manipulation library. It is the absolute standard for working with structured (tabular, multi-dimensional) data in Python.

- **Built on NumPy:** It leverages NumPy's underlying speed but adds row/column labels (indices) for extreme readability.
- **Format Agnostic:** Effortlessly reads and writes from CSV, Excel, SQL databases, JSON, and HDF5.
- **Data Cleansing:** Provides intelligent tools for handling missing data, formatting dates, and merging datasets.

Core Data Structures

Everything in pandas relies on two foundational object types designed to represent structured data.

The Series (1D)

A one-dimensional labeled array. Think of it as a single column in an Excel spreadsheet. It can hold any data type (integers, strings, floats).

```
pd.Series([1, 2, 3])
```

The DataFrame (2D)

A two-dimensional, size-mutable, tabular data structure with labeled axes (rows and columns). It is essentially a dictionary of Series objects.

```
pd.DataFrame(data)
```

The Index

Both Series and DataFrames have an Index (row labels). By default, this is an integer range (0, 1, 2...), but it can be dates, names, or strings.

```
df.index
```

Loading & Viewing Data

1. Reading Data

Pandas instantly transforms raw files into powerful DataFrames.

```
import pandas as pd

# Load a CSV file (replaces standard I/O)
df = pd.read_csv("sales_data.csv")

# Load an Excel file
df_xl = pd.read_excel("report.xlsx")

# Create from a Python Dictionary
data = {'Name': ['Anna', 'Bob'],
        'Age': [28, 34]}
df_dict = pd.DataFrame(data)
```

2. Exploring the DataFrame

Quickly inspect massive datasets without printing everything.

```
# View the top 5 rows
print(df.head())

# View memory usage, columns, & datatypes
print(df.info())

# Statistical summary (mean, min, max, etc)
print(df.describe())

# View all column names
print(df.columns)
```

Selecting & Filtering Data

Indexing & Slicing

Extracting specific columns or rows using `loc` (label-based) and `iloc` (position-based).

```
# Select a single column (returns a Series)
names = df['Name']

# Select multiple columns (returns DataFrame)
subset = df[['Name', 'Salary']]

# Select Rows by Label (Index 'A' to 'C')
df.loc['A':'C', 'Salary']

# Select Rows by Position (Rows 0 to 4)
df.iloc[0:5, :]
```

Boolean Filtering

Querying the dataset by applying conditional logic inside the brackets.

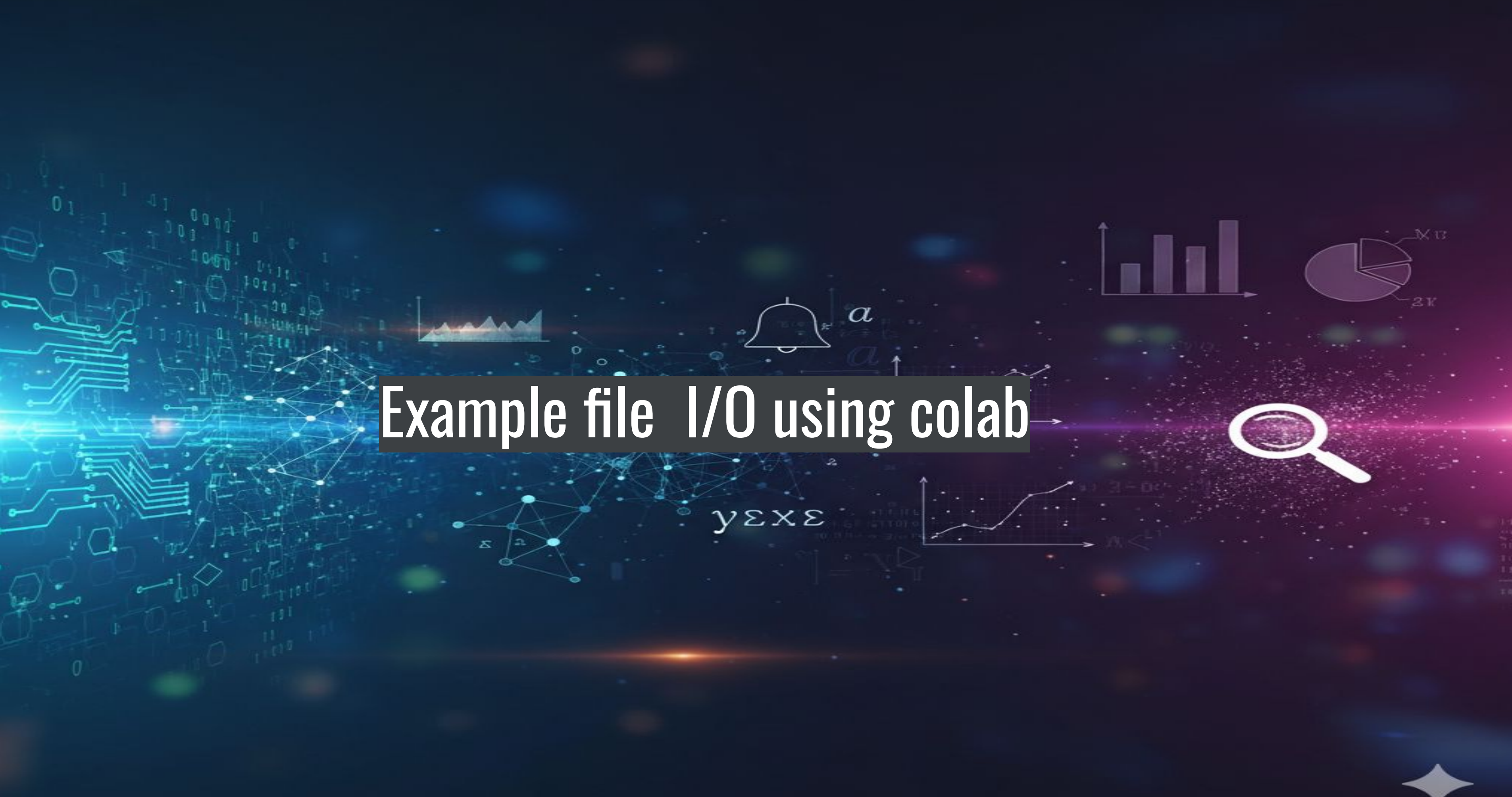
```
# Filter: Find employees older than 30
adults = df[ df['Age'] > 30 ]

# Multiple conditions (AND: & , OR: |)
target = df[(df['Age'] > 30) &
           (df['Dept'] == 'IT')]

# Filter based on a list of values
sales_hr = df[df['Dept'].isin(['Sales', 'HR'])]
```

Data Manipulation

Real-world data is messy.

Pandas provides elegant solutions for handling missing values (NaN), creating new features, and restructuring datasets.

```
# --- Handling Missing Data ---
# Drop rows with any NaN values
clean_df = df.dropna()

# Replace NaN values with 0
filled_df = df.fillna(0)

# --- Feature Engineering ---
# Create a new calculated column
df['Bonus'] = df['Salary'] * 0.10

# Drop an unwanted column
df = df.drop('Old_Column', axis=1)
```



Example file I/O using colab

```python
import matplotlib.pyplot
import numpy
import math

from google.colab import files
files.upload()

f = open("numbers.txt", "r")
```

```python
values = []
i = 0
for l in f:
  i += 1
  valori = l.split()
  if len(valori) == 2:
    values.append(float(valori[0]))
    values.append(float(valori[1]))
  else:
    print ("error while reading line: ", i)
```

```python
m = numpy.mean(values)
s = numpy.std(values)

matplotlib.pyplot.hist(values)
matplotlib.pyplot.title("Istogramma")
matplotlib.pyplot.xlabel("Valore")
matplotlib.pyplot.ylabel("Freq.")

for v in values:
  matplotlib.pyplot.plot(v, 100.0, color = 'green', marker = '.')

matplotlib.pyplot.plot(m, 100.0, color = 'red', marker = 'H')
matplotlib.pyplot.plot(m-2.0*s, 100.0, color = 'red', marker = 'H')
matplotlib.pyplot.plot(m+2.0*s, 100.0, color = 'red', marker = 'H')

matplotlib.pyplot.show()
```

```
import matplotlib.pyplot
impor
impor

from
files

f = o

values
i = 0
for l
  i +=
  valu
  if l
    va
    va
  else
    pr
```

Choose Files No file chosen Upload widget is only available when the cell has been

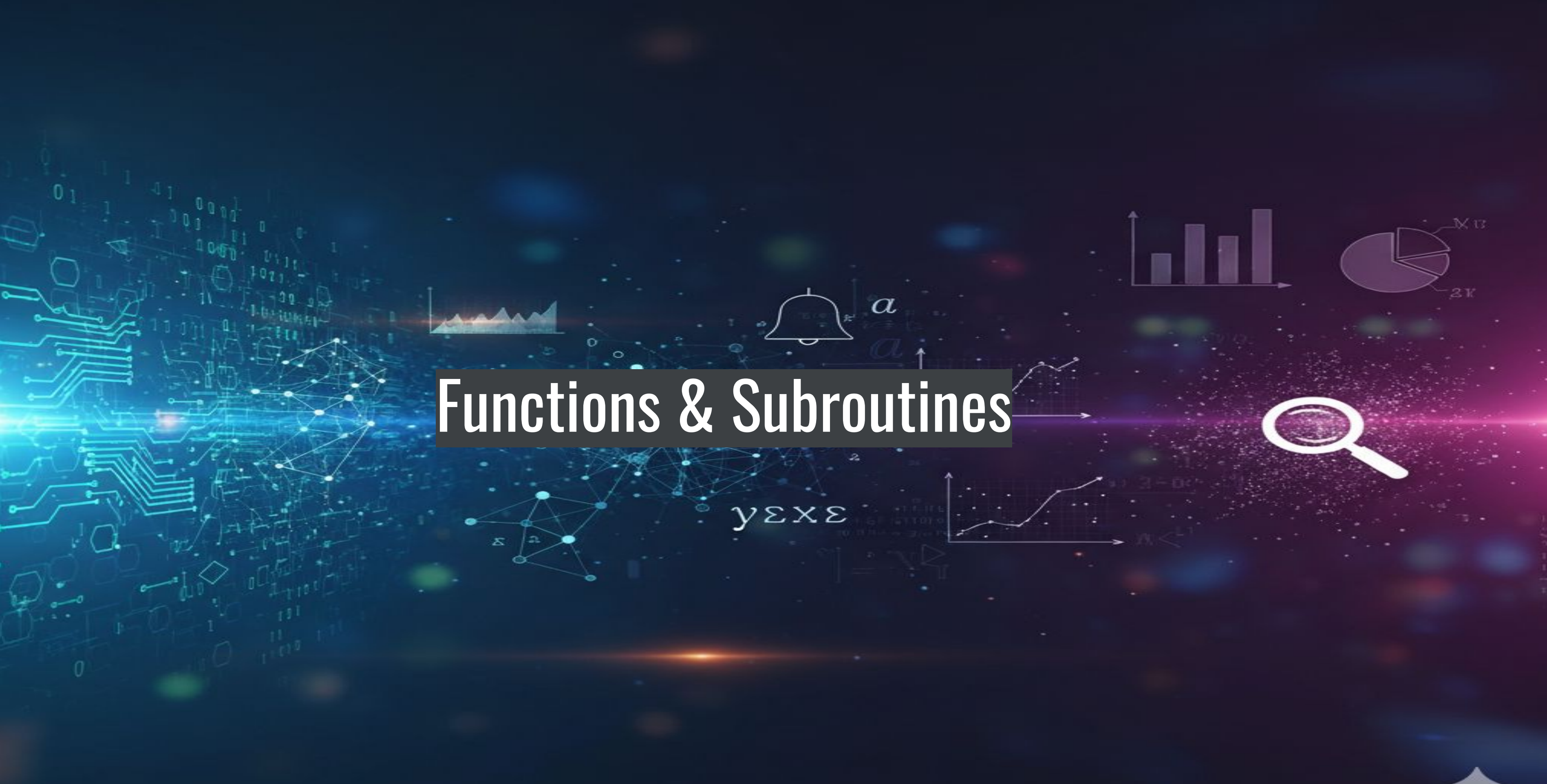
Saving numbers.txt to numbers.txt

Istogramma

Freq.

Valore

Functions & Subroutines



The Python Approach

In many older programming languages, Functions and Subroutines are distinct concepts. Python simplifies this by using a single, unified architecture.

Subroutines

A sequence of instructions designed to **perform a specific action**, like printing to the console, writing to a file, or modifying global state.

In Python: It's just a function that does not possess an explicit return statement. By default, Python silently returns None at the end.

Functions

A sequence of instructions designed to **calculate and return data**. It takes inputs (arguments), processes them, and outputs a new value.

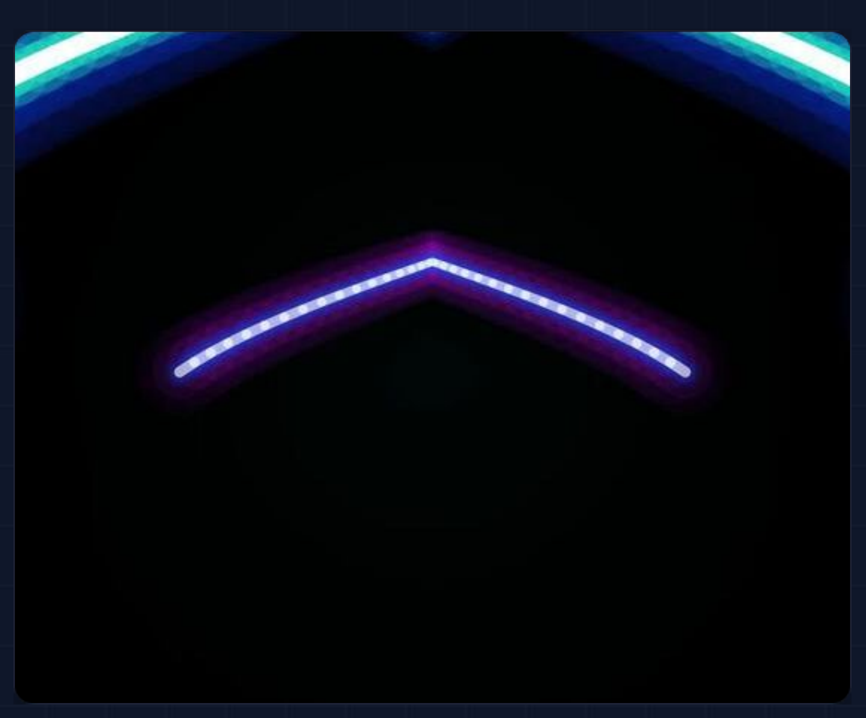
In Python: This uses the exact same syntax, but requires an explicit return keyword to pass data back to the caller.

Anatomy of a Definition

Both functions and subroutines are implemented using the def keyword. They follow a strict indentation block structure.

```
# A typical Python Function
def calculate_area(width, height):
    """Returns the area of a rectangle."""
    area = width * height
    return area

# Calling the function
result = calculate_area(5, 10)
print(result)  # Output: 50
```

- **def**: The keyword indicating a definition.
- **Parameters**: Local variables passed in.
- **Docstring**: Triple-quotes describing the block.



Default Parameter Values

Optional Arguments

You can assign a default value to a parameter using the assignment operator (=) in the function definition. If the caller skips it, Python uses the default.

```
def greet(name, msg="Welcome!"):
    print(f"Hello {name}, {msg}")

# Skipping the second argument:
greet("Alice")
# Hello Alice, Welcome!

# Overriding the default:
greet("Bob", "Good morning!")
# Hello Bob, Good morning!
```

The Golden Rule

In Python, **non-default arguments cannot follow default arguments**. All parameters with default values must be placed at the *end* of the parameter list.

✕SYNTAX ERROR

```
def create_user(role="user", username):
    pass
```

✓CORRECT

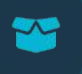
```
def create_user(username, role="user"):
    pass
```

The LEGB Rule

When a variable is called, Python searches for its value by looking outward through four specific layers of scope (Data Locality).

Local (L)

Variables defined *inside* the current function. These are isolated and destroyed when the function finishes running.



Enclosing (E)

Variables defined inside any enclosing functions (relevant when you nest functions inside other functions).

Global (G)

Variables defined at the top-level of a module/script. Accessible from anywhere within that specific file.

Built-in (B)

Pre-assigned names inherent to Python itself (like print, len, or Exception).

Scope in Action

Local Shadowing

If you create a local variable with the same name as a global one, it "shadows" (hides) the global one inside the function.

```
x = 10  # Global Scope

def my_func():
    x = 5  # Local Scope (shadows global)
    print("Inside:", x)

my_func()          # Prints: Inside: 5
print("Outside:", x) # Prints: Outside: 10
```

The global Keyword

To intentionally modify a global variable from *inside* a local function scope, use the global keyword.

```
counter = 0

def increment():
    global counter
    counter += 1

increment()
increment()
print(counter)  # Prints: 2
```

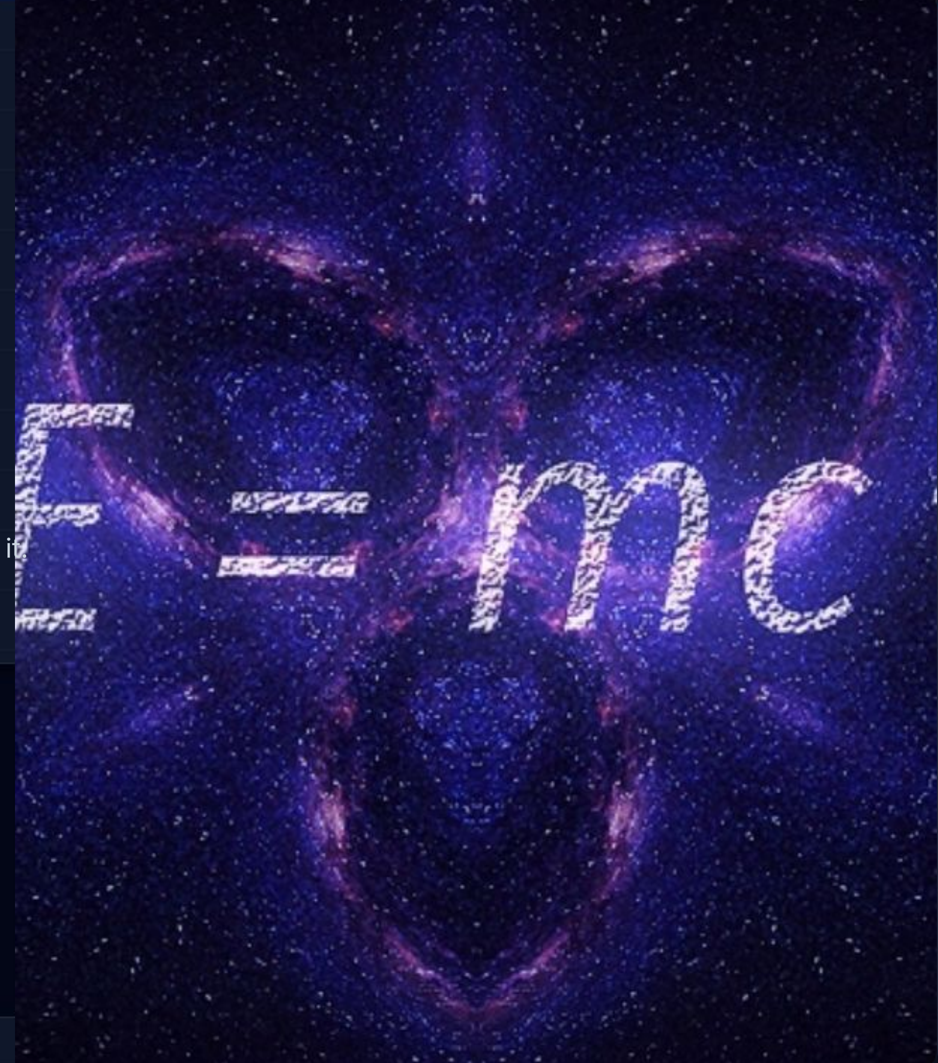
Returning Multiple Values

Python's Secret Weapon: Tuples

Unlike Java or C++, Python functions can seemingly return multiple variables at once separated by commas.

However, under the hood, Python is performing a trick called **Tuple Packing**. It bundles all the requested values into a single immutable Tuple object, returns that one object, and relies on the caller to unpack it.

```python
def get_user_info():
    name = "Alice"
    age = 30
    role = "Admin"

    # Looks like returning 3 things...
    # Actually returning ONE tuple: ("Alice", 30, "Admin")
    return name, age, role
```



Tuple Unpacking

x, y = get()

ASSIGNMENT • EXPANSION

Catching the Output

When calling a function that returns multiple packed values, you can use **Tuple Unpacking** to instantly map the outputs back into individual variables in a single line of code.

```
# Call the function from the previous slide
user_name, user_age, user_role = get_user_info()

print(user_name)  # Prints: Alice
print(user_age)   # Prints: 30

# Rule: The number of variables on the left
# MUST perfectly match the size of the returned tuple.
```

Fibonacci & Recursion

The Fibonacci Sequence

The Fibonacci sequence is a famous series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.

```
0, 1, 1, 2, 3, 5, 8, 13, 21...
```

- **Natural Design:** It describes the spiral arrangement of leaves, pinecones, and even the shape of galaxies.
- **The Golden Ratio:** As the sequence progresses, the ratio of successive numbers approaches exactly 1.618.

The Mathematical Definition

To translate a pattern into code, we must first define it using strict mathematical logic.

The Base Cases

Every sequence needs a starting point. For Fibonacci, we must explicitly define the first two numbers because they have no predecessors to sum together.

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

The Recursive Step

Once the base cases are established, any number in the sequence ( ) can n be calculated by referencing the previous steps.

$$F_n = F_{n-1} + F_{n-2}$$

Enter Recursion

A programming concept where a function solves a problem by calling itself.

What is a Recursive Function?

In computer science, recursion occurs when a function calls itself during its execution. It solves a large problem by breaking it down into smaller, identical problems.

- **The Base Case:** The absolute most critical part. The function must have a condition where it stops calling itself, otherwise it will crash the program (Stack Overflow).
- **The Recursive Case:** The part where the function calls itself, inching slightly closer to the Base Case with each iteration.

The Python Implementation

```python
def fibo(n):
  # 1. The Base Case
  if n == 0 or n == 1:
    return n

  # 2. The Recursive Case
  else:
    return fibo(n - 1) + fibo(n - 2)


# Calculate the 10th number
if __name__ == "__main__":
  print(fibo(10)) # Output: 55
```

Breaking Down the Logic

Notice how perfectly the Python code mirrors the mathematical formula.

- **The `if` statement** acts as our safety net. When the sequence finally trickles down to 0 or 1, it hands the raw number back up the chain without calling the function again.
- **The `else` statement** is where the magic happens. To find fibo(10), it must pause and ask for fibo(9) and fibo(8). It cannot return an answer until those inner queries resolve.

The Call Tree

Visualizing Execution

When you run fibo(4), the computer does not instantly know the answer. It builds a branching tree of function calls.

```
fibo(4) evaluates to:
  fibo(3) + fibo(2)
Which expands to:
  (fibo(2) + fibo(1)) + (fibo(1) + fibo(0))
```

The tree branches outwards until every single tip hits a Base Case (0 or 1), at which point all the numbers bubble back up the tree to be added together.

The Danger of Naive Recursion

The Redundant Calculation Problem

While the recursive code is beautiful and short, it is **dangerously inefficient** for large numbers.

Look back at the call tree. To calculate fibo(4), the computer calculated fibo(2) twice. It recalculated the exact same math independently.

For fibo(10), the function runs about 109 times. But for fibo(50), it will run over **20 Billion times**, likely freezing your computer!

First-Class Functions

Storing callable behaviors inside Python data structures like Dictionaries and Lists to build elegant, dynamic logic.

Functions are Objects

In Python, functions are **first-class objects**. This is a fundamental concept that separates Python from many older, stricter languages.

- **Memory Allocation:** When you write def my_func():, you are creating an object in memory, just like typing x = 5.
- **Variable Assignment:** You can assign a function to a new variable name.
- **Data Structures:** Functions can be packed into lists, passed as arguments, or used as values inside a dictionary.

Step 1: Defining the Behaviors

Before we can store functions, we need to create them. Here, we define four simple, isolated mathematical operations.

÷ Division & Multiplication

```
def divido(a, b):
    return a / b

def moltiplico(a, b):
    return a * b
```

+ Addition & Subtraction

```
def sommo(a, b):
    return a + b

def sottraggo(a, b):
    return a - b
```

Step 2: The Dispatcher Dictionary

The Core Concept

Now we map string commands to our function objects inside a dictionary. This is known as the **Dispatcher Pattern**.

```
# Map string keys to function objects
operazioni = {
    "/" : divido,
    "*" : moltiplico,
    "+" : sommo,
    "-" : sottraggo
}
```

⚠ **Crucial Detail:** Notice we wrote divido, NOT divido(). We are storing a *reference* to the function in memory, we are not executing it yet!

Step 3: Applying the Logic

We can now build a dynamic master function. It accepts a list of values, a scalar, and a string representing the operation we want to perform.

- **Lookup:** operazioni[c] retrieves the correct function object from the dictionary based on the string.
- **Execution:** The (v, d) immediately following it executes that retrieved function with our arguments.

```python
def opera(vals, d=2.0, c="/"):
    res = []
    for v in vals:
        # 1. Lookup: operazioni[c]
        # 2. Execute: (v, d)
        risultato = operazioni[c](v, d)

        res.append(risultato)

    return res
```

Step 4: Execution & Output

The Input Data

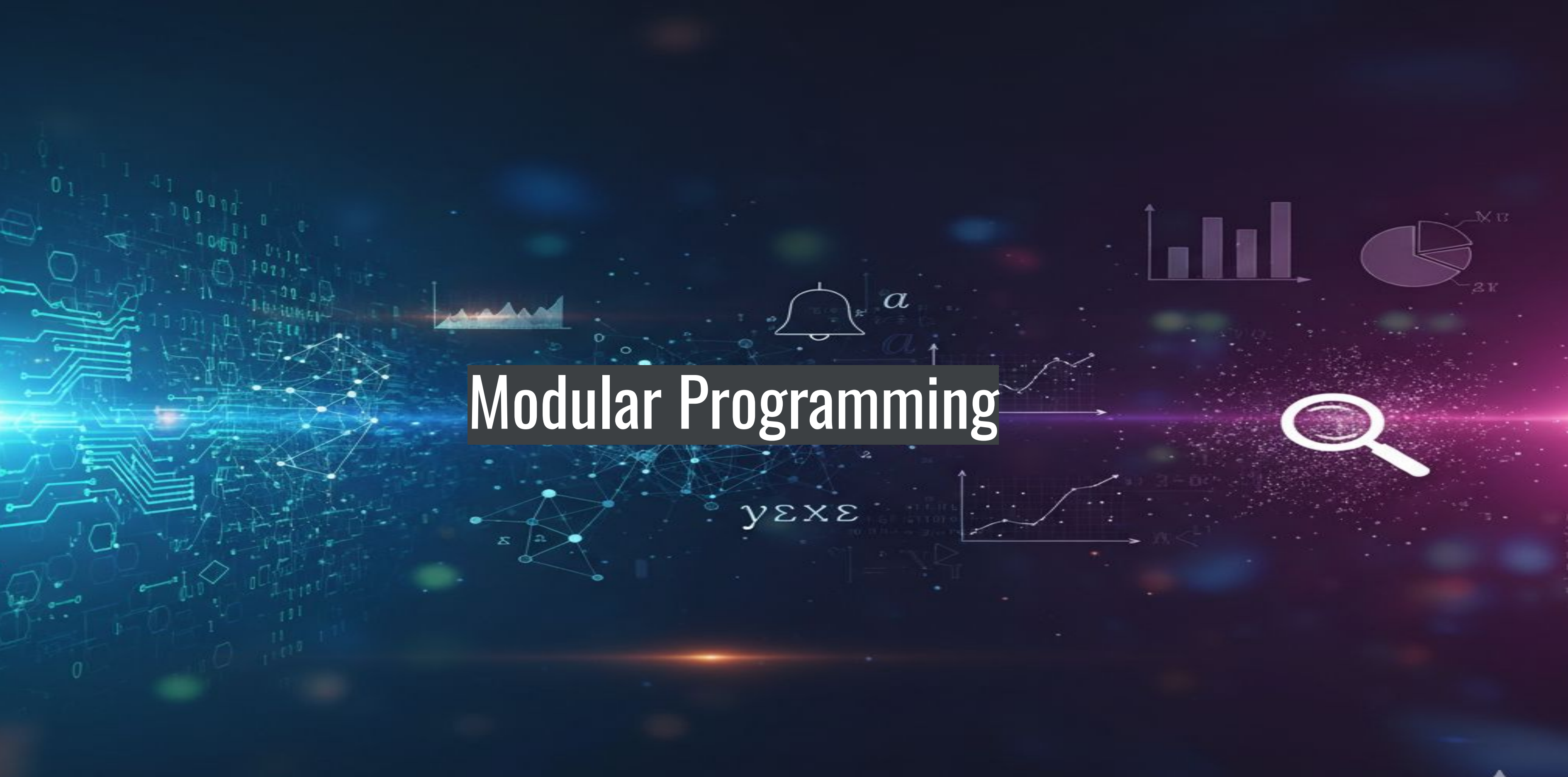
We establish our target array and run the function using its default arguments (c="/" and d=2.0).

```
vals = [1.0, 3.5, 5.6, 7.8]
print(vals)
# [1.0, 3.5, 5.6, 7.8]


print(opera(vals))
# [0.5, 1.75, 2.8, 3.9]
```

Dynamic Overrides

By passing different arguments, we seamlessly hot-swap the underlying mathematical behavior.

```
# Override the divisor (d = 3.0)
print(opera(vals, 3.0))
# [0.333, 1.166, 1.866, 2.6]

# Override the operation (c = "+")
print(opera(vals, c="+"))
# [3.0, 5.5, 7.6, 9.8]
```



Modular Programming

What is a Module?

A module is simply a file containing Python statements and definitions. It is the building block of large-scale applications.

- **Reusability:** Write complex logic once, and reuse it across dozens of different scripts.
- **Organization:** Break massive monolithic programs into small, logical, manageable files.
- **Namespace Protection:** Keep variables and function names isolated so they don't accidentally clash with code in other files.

Creating and Using a Module

Creating a module requires no special syntax. Simply save your Python code in a file with a .py extension.

1. The Definition

Create a file named math_tools.py.

```python
# File: math_tools.py

def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

PI = 3.14159
```

2. The Execution

In a separate file (e.g., main.py), load the module using the import keyword.

```python
# File: main.py
import math_tools

# Access elements using "dot notation"
result = math_tools.add(5, 3)
print(result)  # Output: 8

print(math_tools.PI) # Output: 3.14159
```

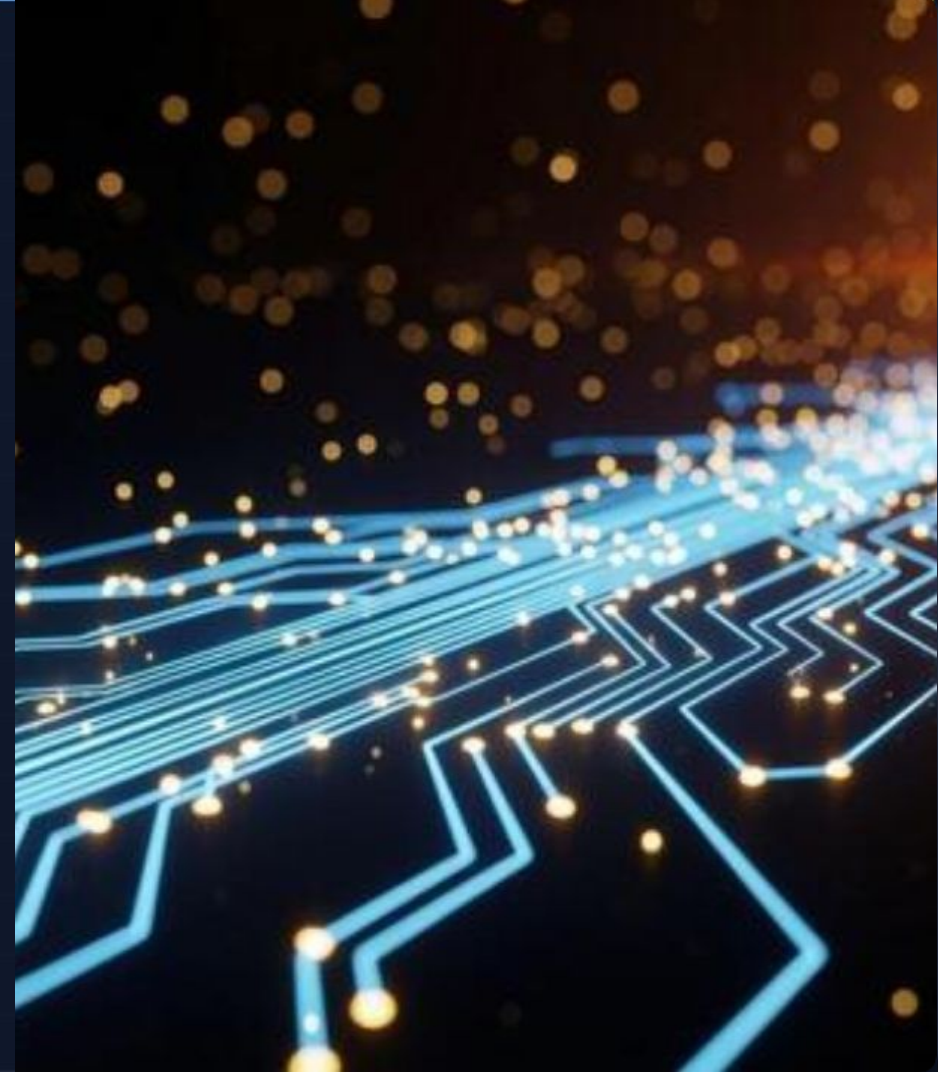
Importing Specific Elements

The from ... import ... Statement

Often, you don't need an entire module. You only need one specific function or variable.

By specifying exactly what to import, you bring those elements directly into your current namespace. This eliminates the need to use "dot notation" when calling them.

```
# Extract exactly what we need
from math_tools import add, PI

# Call 'add' directly without 'math_tools.'
result = add(10, 5)
print(result)  # Output: 15

print(PI)      # Output: 3.14159
```



Aliasing: The as Keyword

You can rename modules and elements at the time of import. This is crucial for avoiding name collisions or shortening lengthy module names.

Aliasing Whole Modules

Standard practice in data science is to abbreviate heavy library names to keep code clean.

```
# Rename the module to 'pd'
import pandas as pd
import numpy as np

# Use the alias for dot notation
df = pd.DataFrame()
arr = np.array([1, 2])
```

Aliasing Specific Elements

If you import a function that shares a name with a variable you already have, alias it to prevent a crash.

```
# Assume we already have an 'add' function
def add(a, b):
    print("Local function")

# Alias the imported function to resolve confl
from math_tools import add as math_add

result = math_add(5, 5)
```

How Python Finds Modules

When you execute import my_module, Python searches for that file through a specific, ordered list of directories known as sys.path.

1. **Current Directory:** Python first looks in the exact folder where your running script is located.
2. **PYTHONPATH:** It then checks directories defined in your OS environment variables.
3. **Standard Library:** Next, it checks the built-in installation directory.
4. **Site-Packages:** Finally, it checks where third-party packages (installed via pip) live.

The Module Ecosystem

Standard Library

Python's "Batteries Included" philosophy. Hundreds of powerful modules (like math, os, datetime, json) are pre-installed and ready to import instantly.

Third-Party

Over 400,000 packages exist on PyPI. You download them via the terminal (pip install requests), then import them into your code just like standard modules.

Custom Modules

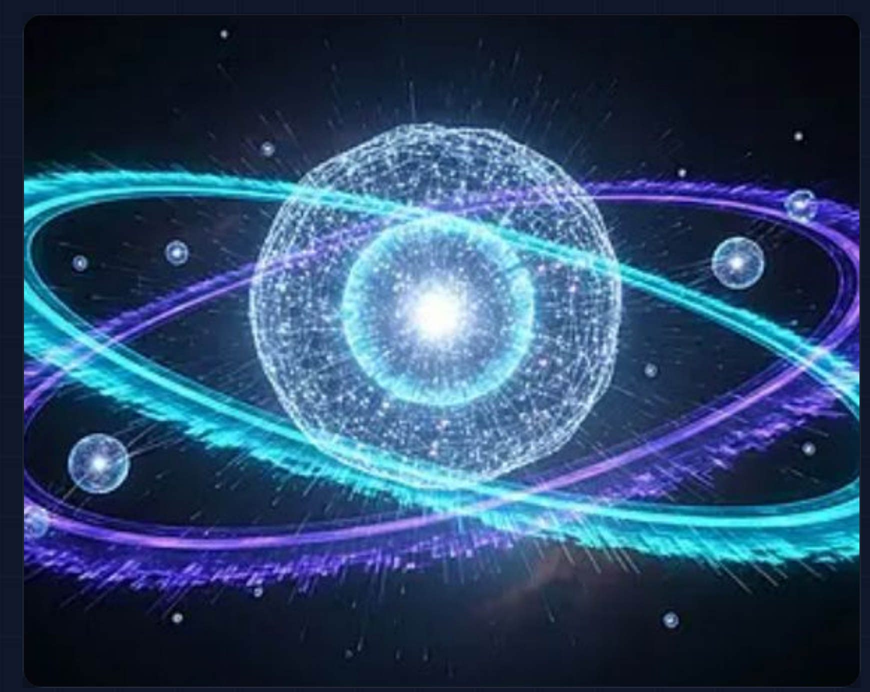
The files you write yourself. Grouping related functions into your own custom modules is the first step toward building professional, scalable applications.

Python Classes & Objects

What is a Class?

A class is a blueprint for creating objects. It defines the structure and behavior of a concept before any real instances of that concept exist in memory.

- **The Blueprint (Class):** The generic definition of an "Atom". It dictates that an atom *will have* a symbol, coordinates, and a color.
- **The Instance (Object):** A specific, tangible version of that class built in memory. (e.g., A specific Oxygen atom at coordinates X, Y, Z).
- **Encapsulation:** Classes bundle data (Variables/Attributes) and behavior (Functions/Methods) into one single entity.



The Blueprint: class atom

The foundation of a Python class is the constructor. It tells Python exactly how to build a new object when requested.

The __init__ Method

```
class atom(object):
  # The Constructor
  def __init__(self, at, x, y, z):
    # Setting internal attributes
    self.simbolo = at
    self.coordinate = (x, y, z)
    self.raggio = 0.0
    self.colore = (1.0, 1.0, 1.0)
```

The Magic of self

The __init__ function is automatically called when you create a new atom. It accepts parameters like at, x, y.

The self variable is critical. It refers to the *specific instance* being created. By typing self.simbolo = at, we are attaching the provided symbol directly to that individual object's memory space.

Methods & Encapsulation

Objects shouldn't just hold data; they should manage how that data is accessed and modified. This is done through "Methods" (functions inside a class).

Getter Methods

Methods like get_radius(self) provide safe, read-only access to an object's internal state without exposing the raw variables directly.

```
def get_radius(self):
    return self.raggio
```

Setter Methods

Methods like set_color(self, r, g, b) allow controlled updates to internal data. In production, these often include validation checks.

```
def set_color(self, r, g, b):
    self.colore = (r, g, b)
```

The self Link

Notice every method requires self as the first parameter. This guarantees the method is modifying *this* specific atom, not another atom elsewhere in the code.

Overloading Built-in Behavior

By default, if you type print(my_atom), Python will print a generic, ugly memory address like <__main__.atom object at 0x7f...>.

We can override this behavior using **Magic Methods** (Dunder methods), specifically __repr__ (Representation) or __str__.

> In the code provided, __repr__ intercepts the print command and calls the custom get_str() method to output beautifully formatted text instead.

```
# Custom string formatting method
def get_str(self):
  return '%s %10.4f %10.4f %10.4f'  % (
      self.simbolo,
      self.coordinate[0],
      self.coordinate[1],
      self.coordinate[2]
  )

# Magic method: Overloads printing
def __repr__(self):
  return self.get_str()
```

Composition: class molecule

Building Complexity

Object-Oriented Programming shines when we combine simple objects to create complex ones. This is called **Composition**. A Molecule is simply an object that *contains* a list of Atom objects.

```
class molecule(object):
  def __init__(self, nome="noname"):
    self.name = nome
    # The list holding our atom objects!
    self.lista_atomi = []

  def add_atom(self, atom):
    self.lista_atomi.append(atom)
```

When printing the molecule, its __repr__ method uses a loop to ask every atom inside its list to print itself!

Executing the Physics

Putting it all together

We instantiate our objects, build the relationships, and print the final structure.

```
# 1. Instantiate the Atom Objects (The Blueprint becomes reality)
h1 = atom('H', 0.0, 0.0, 0.0)
h2 = atom('H', 0.0, 0.0, 1.0)
o  = atom('O', 0.0, 1.0, 0.0)

# 2. Instantiate the Molecule Object
acqua = molecule("H2O")

# 3. Add the Atoms to the Molecule (Composition)
acqua.add_atom(h1)
acqua.add_atom(h2)
acqua.add_atom(o)

# 4. Watch the overloaded __repr__ methods do the formatting
print(acqua)
```

Putting it to Work: Methane (CH_4)

Importing our mol.py module and building the molecule:

```
import mol

# 1. Create the molecule
m = mol.molecule("metano")

# 2. Add Carbon and Hydrogen atoms
m.add_atom(mol.atom("C", 3.875, 0.678, -8.417))
m.add_atom(mol.atom("H", 3.800, 1.690, -8.076))
m.add_atom(mol.atom("H", 4.907, 0.410, -8.516))
m.add_atom(mol.atom("H", 3.406, 0.026, -7.711))
m.add_atom(mol.atom("H", 3.389, 0.583, -9.366))

# 3. Print the resulting object
print(m)

# Note: m1 = m creates a reference, not a copy!
# Modifying m1 will modify m.
```

The resulting **Terminal Output**:

```
Molecule metano
ha 5 atomi
C      3.8750     0.6780     -8.4170
H      3.8000     1.6900     -8.0760
H      4.9070     0.4100     -8.5160
H      3.4060     0.0260     -7.7110
H      3.3890     0.5830     -9.3660
```

The Magic of __repr__: Notice how calling print(m) automatically cascades. The Molecule formats its header, then loops through its internal list, asking each Atom object to format its own coordinates!

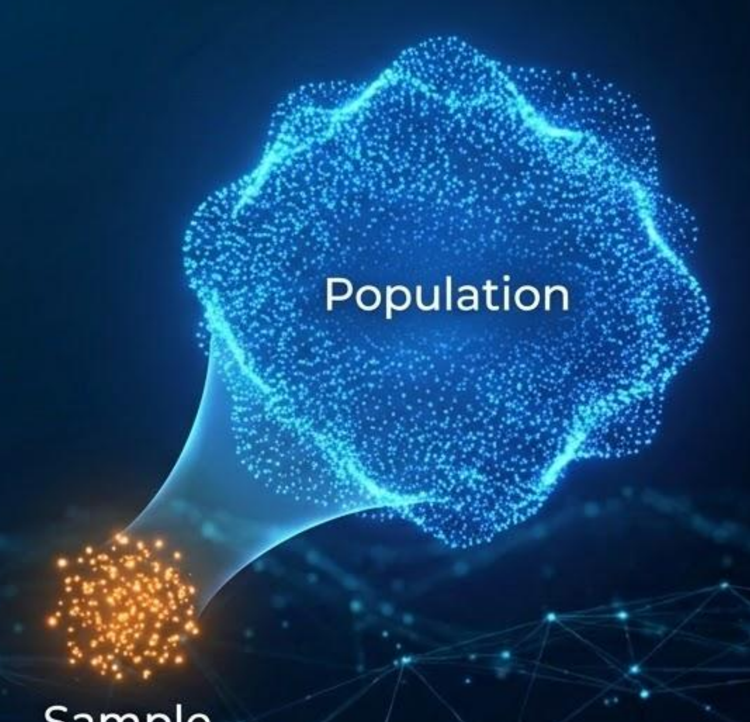
MONTECARLO APPROACH

MONTE CARLO METHODS

In the nuclear expressions, a fast particle fully hits the nucleus of an atom. This shatters into many particles, which hit the nuclei of other neighboring atoms, which are shattered in turn, according to a chain reaction, during which a great deal of energy is released. The process will last until it involves the whole universe or will it stop after a certain number of reactions?

THE CORE DEFINITION

The Monte Carlo method consists in seeking the solution of a problem, representing it as a parameter of a hypothetical population and in estimating this parameter by examining a sample of the population obtained through sequences of random numbers.



The Mathematical Engine

Why does generating random numbers yield highly precise deterministic answers? The method relies on two foundational pillars of probability theory.

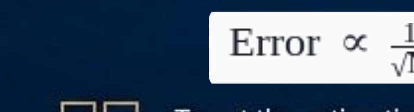
The Law of Large Numbers

As the size of a sample increases, the average (mean) of the sample will continually approach the expected value of the entire population.

In Monte Carlo terms: Generating 10 random samples provides a rough guess, but generating 1,000,000 random samples guarantees a highly precise estimation of the true parameter.

The Standard Error

The accuracy of a Monte Carlo estimate improves in proportion to the **square root** of the number of samples.



$$\text{Error} \propto \frac{1}{\sqrt{N}}$$

To cut the estimation error in half, we must quadruple the amount of random samples generated.

The 4-Step Methodology

1. **Define the Domain**
   Establish the mathematical bounds of possible inputs for the problem (the "hypothetical population").

2. **Generate Random Inputs**
   Create a vast sequence of random numbers drawn precisely from the defined probability distribution over the domain.

3. **Perform Computation**
   Execute a deterministic calculation on every single randomly generated input (e.g., testing if a point falls inside a geometric shape).

4. **Aggregate the Results**
   Combine all the individual calculations into a final statistical tally to estimate the unknown parameter.

Visualizing Convergence

As the sequence of random numbers grows, the statistical estimate fluctuates less wildly and securely converges upon the true deterministic parameter of the population.

Broad Industry Applications

Quantitative Finance

Used extensively to model the unpredictable behavior of financial markets, assessing portfolio risks, and pricing complex derivative options under immense uncertainty.

Physical Sciences

Essential in particle physics for modeling radiation transport, simulating the behavior of subatomic particles, and predicting quantum mechanical fluid dynamics.

Engineering Reliability

Engineers apply Monte Carlo simulations to test the structural integrity of bridges and aircraft by subjecting mathematical models to millions of random, simulated stress factors.

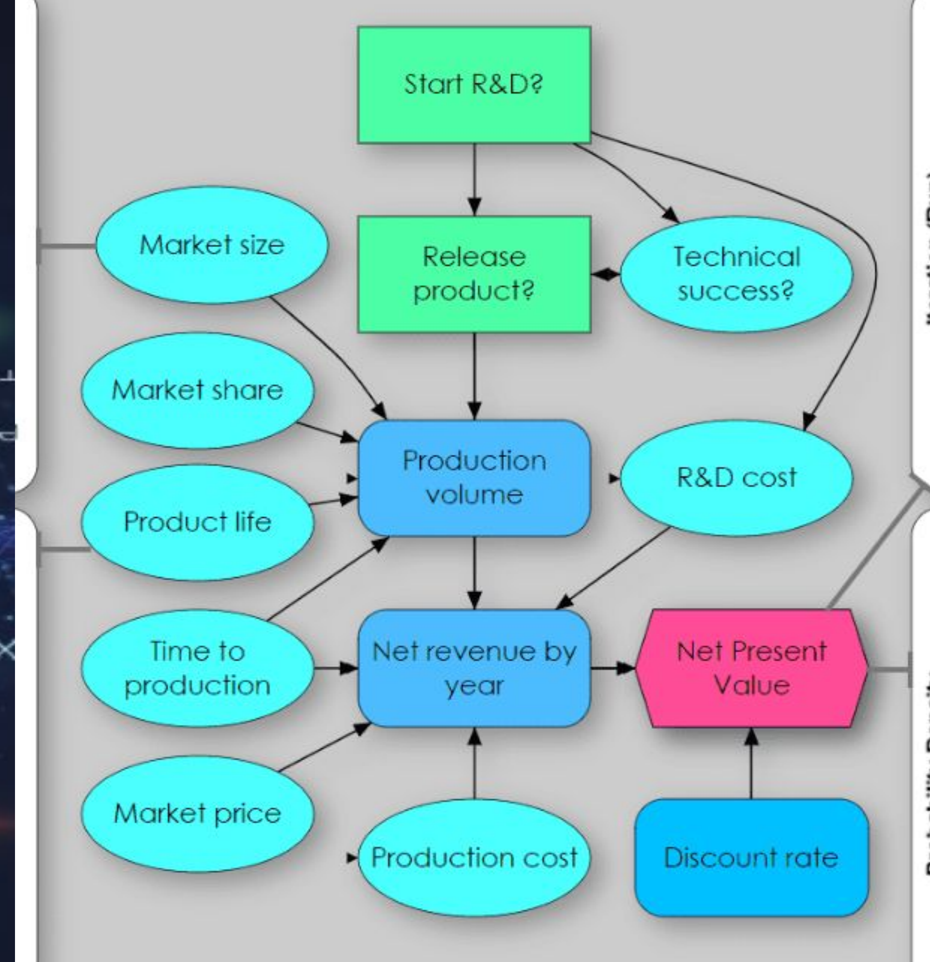
The Power of Randomness

Transforming Calculus into Arithmetic.

The brilliance of the Monte Carlo approach is its ability to bypass impossibly complex analytical mathematics.

Instead of attempting to calculate exact, multi-dimensional deterministic equations—which can take supercomputers years to solve—we simply harness raw computational speed to throw millions of random numbers at the problem.

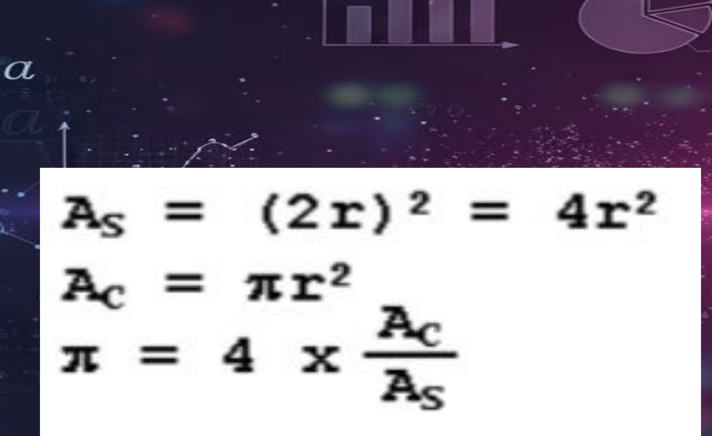
By shifting our perspective from absolute calculation to **statistical estimation**, we unlock rapid, highly accurate solutions across every field of modern science.



MONTECARLO PI ESTIMATION

Compute PI using MC methos

O
r = 1
A



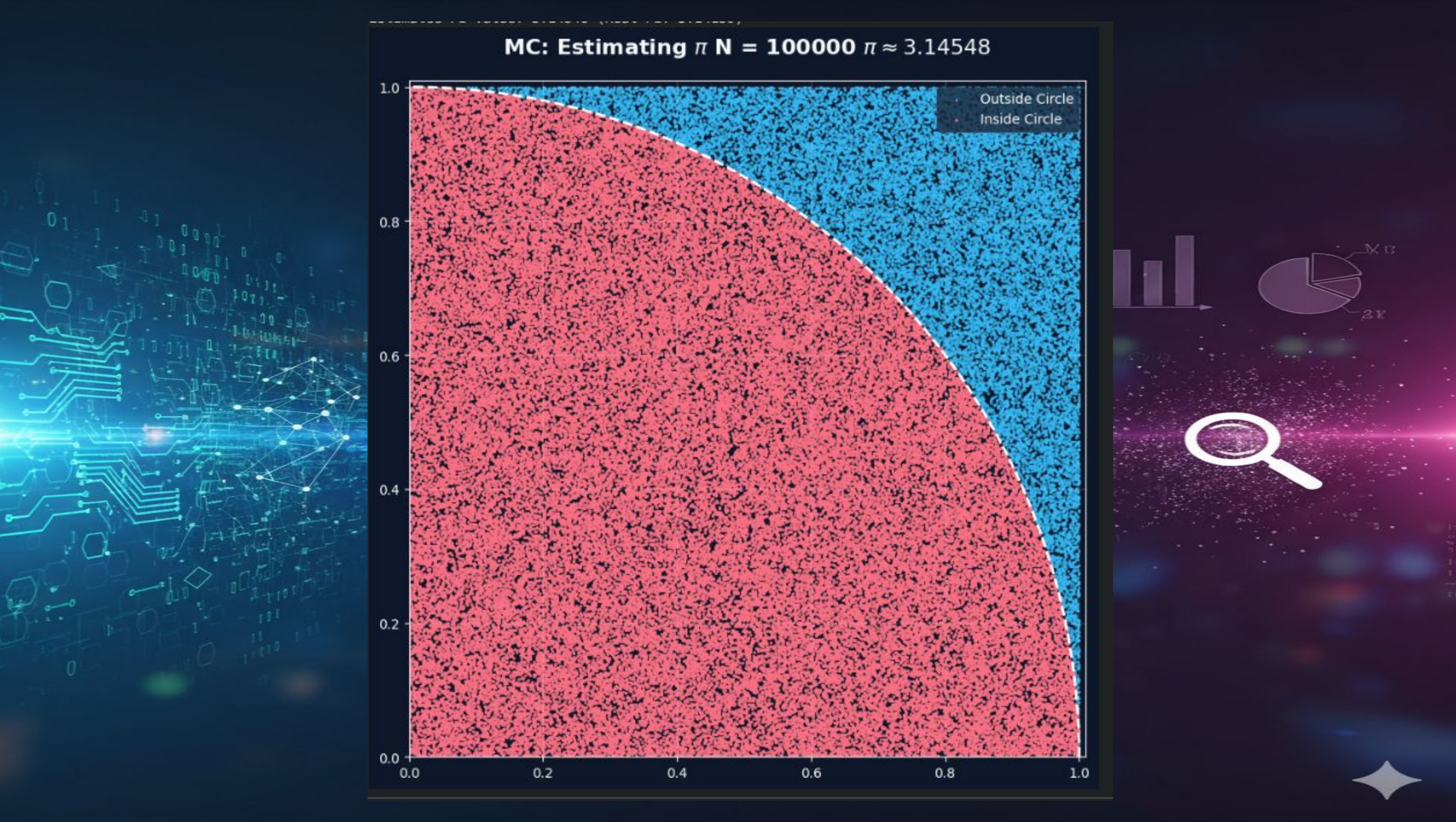
```
import matplotlib.pyplot as plt
import numpy as np
import random

N = 100000

x_inside = []
y_inside = []
x_outside = []
y_outside = []

for i in range(N):
  x = random.uniform(0.0, 1.0)
  y = random.uniform(0.0, 1.0)
  if x**2 + y**2 <= 1:
    x_inside.append(x)
    y_inside.append(y)
  else:
    x_outside.append(x)
    y_outside.append(y)
```

```
pi_estimate = 4 * len(x_inside) / N
```

```
Simulation Complete.
Total Points: 100000
Points Inside (Hits): 7851
Estimated Pi value: 3.14548 (Real Pi: 3.14159)
```

```python
plt.style.use('dark_background')
fig, ax = plt.subplots(figsize=(8, 8))
fig.patch.set_facecolor('#0f172a') # Slate 900 background
ax.set_facecolor('#0f172a')

# Plot points OUTSIDE the circle (Sky Blue)
ax.scatter(x_outside, y_outside, color='#38bdf8', s=2, alpha=0.6, label='Outside Circle')

# Plot points INSIDE the circle (Rose/Red)
ax.scatter(x_inside, y_inside, color='#fb7185', s=2, alpha=0.8, label='Inside Circle')

# Draw the mathematical boundary line of the quarter circle
theta = np.linspace(0, np.pi/2, 100)
ax.plot(np.cos(theta), np.sin(theta), color='white', linewidth=2, linestyle='--')
```

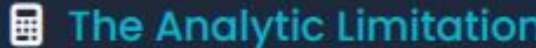
```python
# Formatting the plot
ax.set_aspect('equal')
ax.set_xlim(0, 1.01)
ax.set_ylim(0, 1.01)
ax.set_title(f'MC: Estimating $\pi$ N = {N} $\pi \\approx {pi_estimate:.5f}$',
             fontsize=16, fontweight='bold', pad=20, color='#f8fafc')
ax.legend(loc='upper right', facecolor='#1e293b', edgecolor='#334155')
ax.grid(True, linestyle='-', alpha=0.2)

# Save the generated visualization
plt.tight_layout()
plt.savefig('monte_carlo_pi.png')
print(f"Simulation Complete.")
print(f"Total Points: {N}")
print(f"Points Inside (Hits): {hits}")
print(f"Estimated Pi value: {pi_estimate} (Real Pi: {np.pi:.5f})")
```



MC: Estimating π N = 100000 $\pi \approx 3.14548$
Outside Circle
Inside Circle
0.0
0.2
0.4
0.6
0.8
1.0

MONTECARLO TO COMPUTE AN INTEGRAL

The Calculus Problem



In traditional calculus, calculating the definite integral of a function provides the exact area under its curve.

However, many complex integrals found in physics, finance, and machine learning cannot be solved analytically. Standard numerical methods (like the Trapezoidal rule) also break down or become infinitely slow as the number of variables (dimensions) increases.

The Statistical Solution

The Monte Carlo approach bypasses deterministic calculus entirely.

By leveraging the **Law of Large Numbers** and high-speed computer processors, we can use massive sequences of random numbers to *estimate* the integral with incredibly high precision. There are two primary techniques to achieve this.

Method 1: Average Value (Statistical)

The Mean Value Theorem for Integrals states that a definite integral can be expressed as the width of the interval (b - a) multiplied by the average value of the function over that interval.

$$\mathbb{E}[f(X)] = \int_a^b f(x) \cdot p(x)dx = \frac{1}{b-a}\int_a^b f(x)dx$$

$$\int_a^b f(x)dx = (b-a) \cdot \mathbb{E}[f(X)]$$

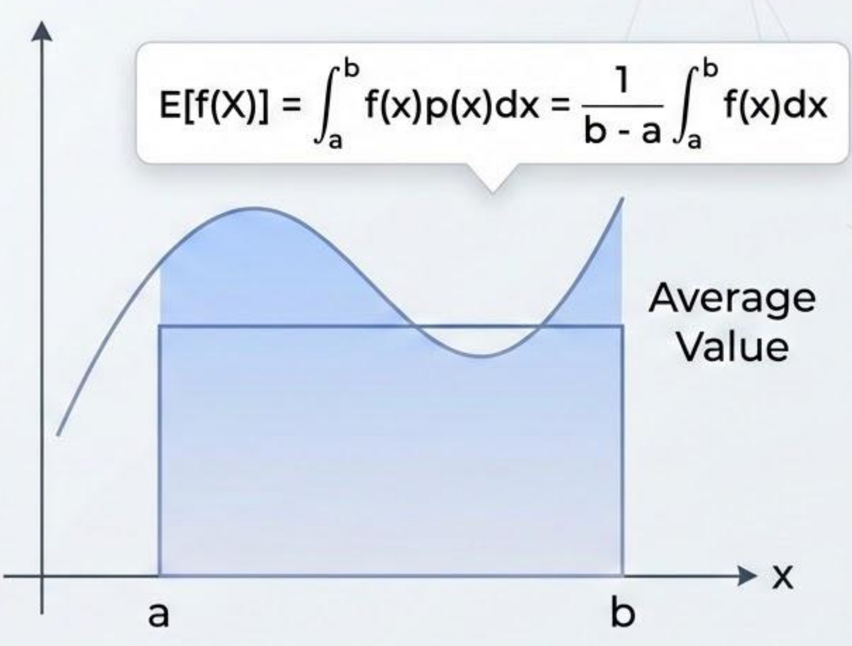
Method 1: Average Value (Statistical)

- This is the most common and robust way to perform Monte Carlo integration. It relies on the statistical concept of the Expected Value.
- The Mean Value Theorem for Integrals states that a definite integral can be expressed as the width of the interval (b - a) multiplied by the average value of the function over that interval.

Method 1: Average Value (Statistical)

If we pick a random variable X that is uniformly distributed over the interval [a, b], its probability density function is $p(x) = 1/\{b-a\}$.

The expected value of our function f(X) is:



Method 1: Average Value (Statistical)

If we pick a random variable X that is uniformly distributed over the interval [a, b], its probability density function is p(x) = 1/{b-a}. The expected value of our function f(X) is:

$$\mathbb{E}[f(X)] = \int_a^b f(x) \cdot p(x) dx = \frac{1}{b-a} \int_a^b f(x) dx$$

$$\int_a^b f(x) dx = (b-a) \cdot \mathbb{E}[f(X)]$$

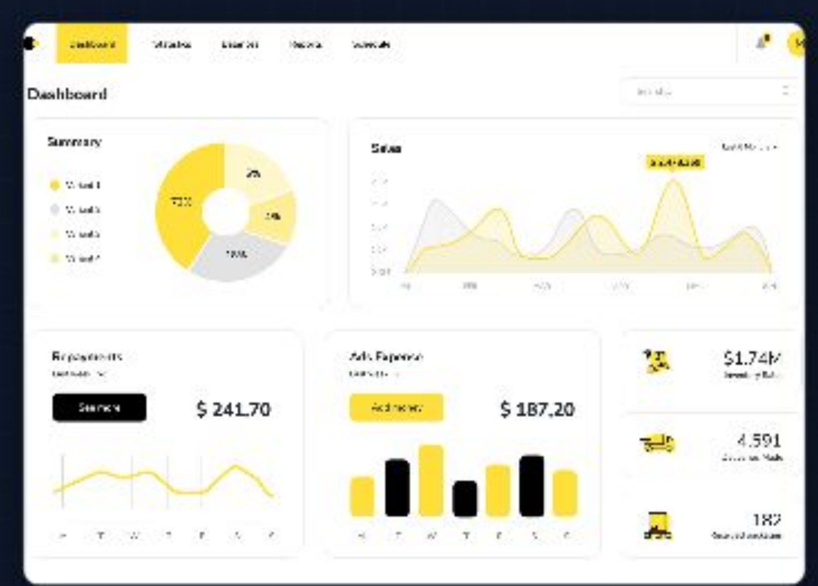
Method 1: Average Value (Statistical)

This is the most robust method for Monte Carlo integration. It relies on the **Mean Value Theorem for Integrals**.

→ **The Logic:** An integral can be expressed as the width of the interval multiplied by the *average height* of the function.

→ **The Execution:** We generate thousands of random points (x_i) across the interval, evaluate the function at each point, and calculate the average.

$$\int_a^b f(x)\,dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i)$$

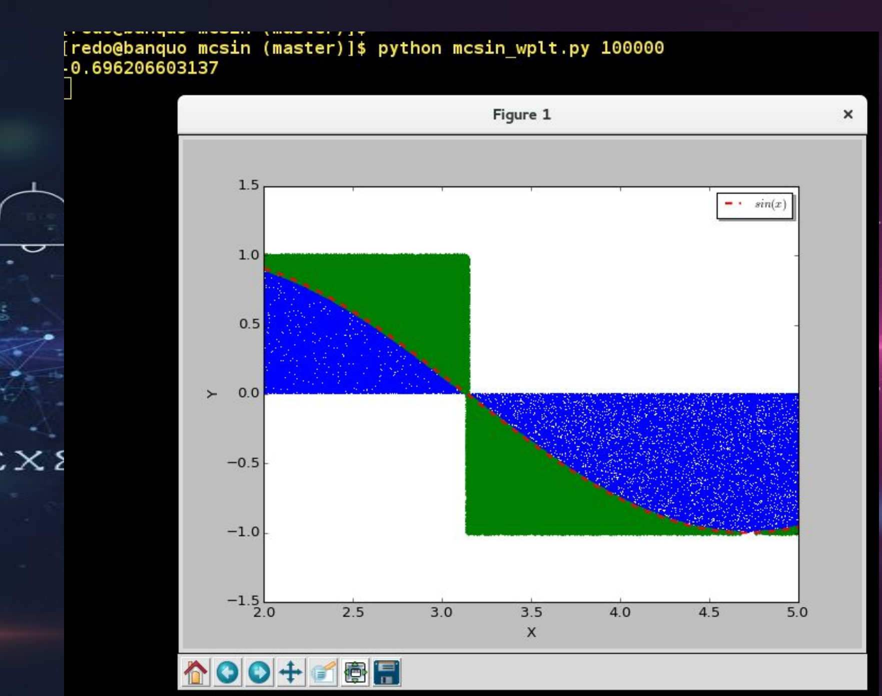


Method 1: Average Value (Statistical)

This is a purely geometric approach, identical to how we estimated the value of Pi (π) with a circle inside a square.

- **The Box:** Draw a bounding box around the entire function from $x=a$ to $x=b$, and $y=0$ to the maximum height M.
- **The Darts:** Generate random 2D coordinates (x, y) inside this box.
- **The Hits:** Count how many random points fall *under* the curve (where $y < f(x)$).

$$\text{Area} \approx \text{Area}_{\text{box}} \times \frac{\text{Hits}}{\text{Total N}}$$



Method 1: Average Value (Statistical)

Estimating the integral of $f(x) = x^2$ from 0 to 2. (Exact Answer: ~2.666)

1. Average Value Method

```
import numpy as np

# N random x-values between 0 and 2
x = np.random.uniform(0, 2, 1000000)

# Evaluate f(x) and find the average
f_avg = np.mean(x**2)

# Multiply by interval width (b - a)
integral = (2 - 0) * f_avg

print(f"Avg Method: {integral}")
```

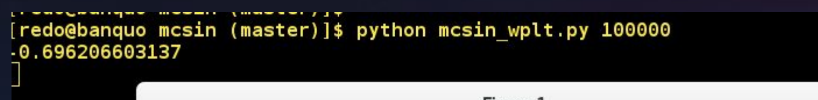
2. Hit-or-Miss Method

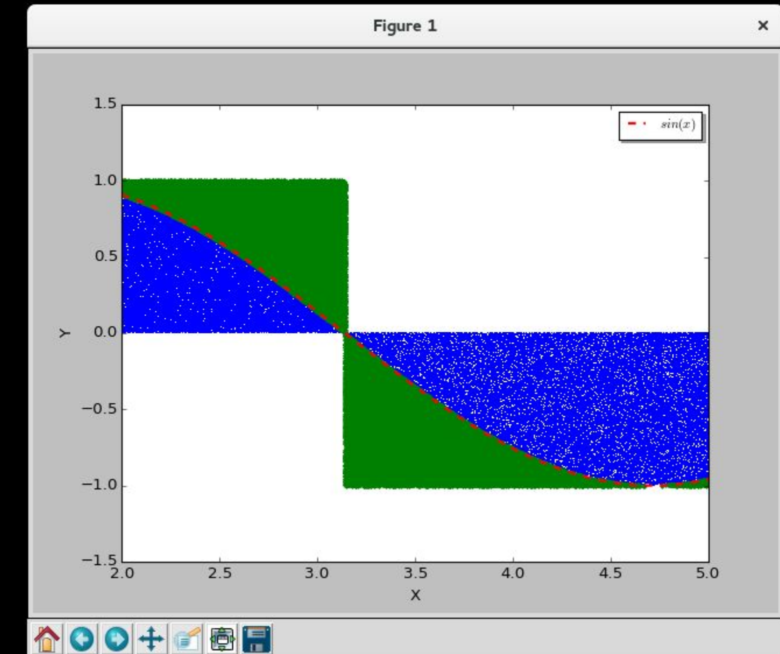
```
import numpy as np

M = 4 # Max height of x² in [0, 2]
x = np.random.uniform(0, 2, 1000000)
y = np.random.uniform(0, M, 1000000)

# Count hits under the curve
hits = np.sum(y < x**2)

# Box Area * (Hits / N)
integral = (2 * M) * (hits / 1000000)

print(f"Hit Method: {integral}")
```

Exercise

```
Set rect_count to 0, xmin to 2.0, xmax to 5.0, ymin to -1.0 and ymax to 1.0

for i from 0 to N :
    Generate x random between xmin and xmax
    Generate y random between ymin and ymax
    if sin(x) > 0.0 and (y <= sin(x) and (y >= 0.0):
        rect_count = rect_count + 1
    else if sin(x) < 0.0 and y >= sin(x) and y <= 0.0
        rect_count = rect_count – 1

integral is equal to (xmax - xmin) * (ymax - ymin) * (rect_count / N)
```

```
[redo@banquo mcsin (master)]$ time python mcsin.py 100
-0.18

real    0m0.013s
user    0m0.010s
sys     0m0.003s
[redo@banquo mcsin (master)]$ time python mcsin.py 1000
-0.726

real    0m0.017s
user    0m0.014s
sys     0m0.003s
[redo@banquo mcsin (master)]$ time python mcsin.py 10000
-0.6834

real    0m0.049s
user    0m0.040s
sys     0m0.008s
[redo@banquo mcsin (master)]$ time python mcsin.py 100000
-0.70938

real    0m0.115s
user    0m0.108s
sys     0m0.007s
[redo@banquo mcsin (master)]$ time python mcsin.py 1000000
-0.703062

real    0m1.049s
user    0m1.021s
sys     0m0.027s
[redo@banquo mcsin (master)]$ time python mcsin.py 10000000
-0.7001124

real    0m10.226s
user    0m10.105s
sys     0m0.116s
[redo@banquo mcsin (master)]$
```





```python
import random
import numpy
import math
import sys

import matplotlib.pyplot as plt

DIM = 10000

p_rect_count = 0
n_rect_count = 0

xmin = 2.0
xmax = 5.0
ymin = -1.0
ymax = 1.0

plt.clf()

xv = []
yv = []
xvo = []
yvo = []
```

```
for i in range(0,DIM):

   x = random.uniform(xmin, math.pi)
   y = random.uniform(0.0, ymax)

   if (y <= math.sin(x)):
     p_rect_count = p_rect_count + 1
     xv.append(x)
     yv.append(y)
   else:
     xvo.append(x)
     yvo.append(y)
```

```
for i in range(0,DIM):

   x = random.uniform(math.pi, xmax)
   y = random.uniform(ymin, 0.0)

   if (y >= math.sin(x)):
     n_rect_count = n_rect_count + 1
     xv.append(x)
     yv.append(y)
   else:
     xvo.append(x)
     yvo.append(y)
```

```
p = 1.0 * ((math.pi - xmin) * ( float(p_rect_count) / float(DIM) ))
n = -1.0 * ((xmax - math.pi) * ( float(n_rect_count) / float(DIM) ))

print(p + n)

plt.plot(xv, yv, ',')
plt.plot(xvo, yvo, '.')
x = numpy.linspace(2.0, 5.0, 1000)
f = numpy.sin(x)
#print (x)
#print (f)
plt.plot(x, f, 'red', linestyle='--', linewidth=2, label='$sin(x)$')
plt.legend(loc='upper right', shadow=True, fontsize='small')

#axes = plt.gca()
#axes.set_xlim([2.0, 5.0])
#axes.set_ylim([-1.5, 1.5])

plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

EXERCISE

Exercise

Write a program to represent the cos function in the interval -5.0 < x < 5.0