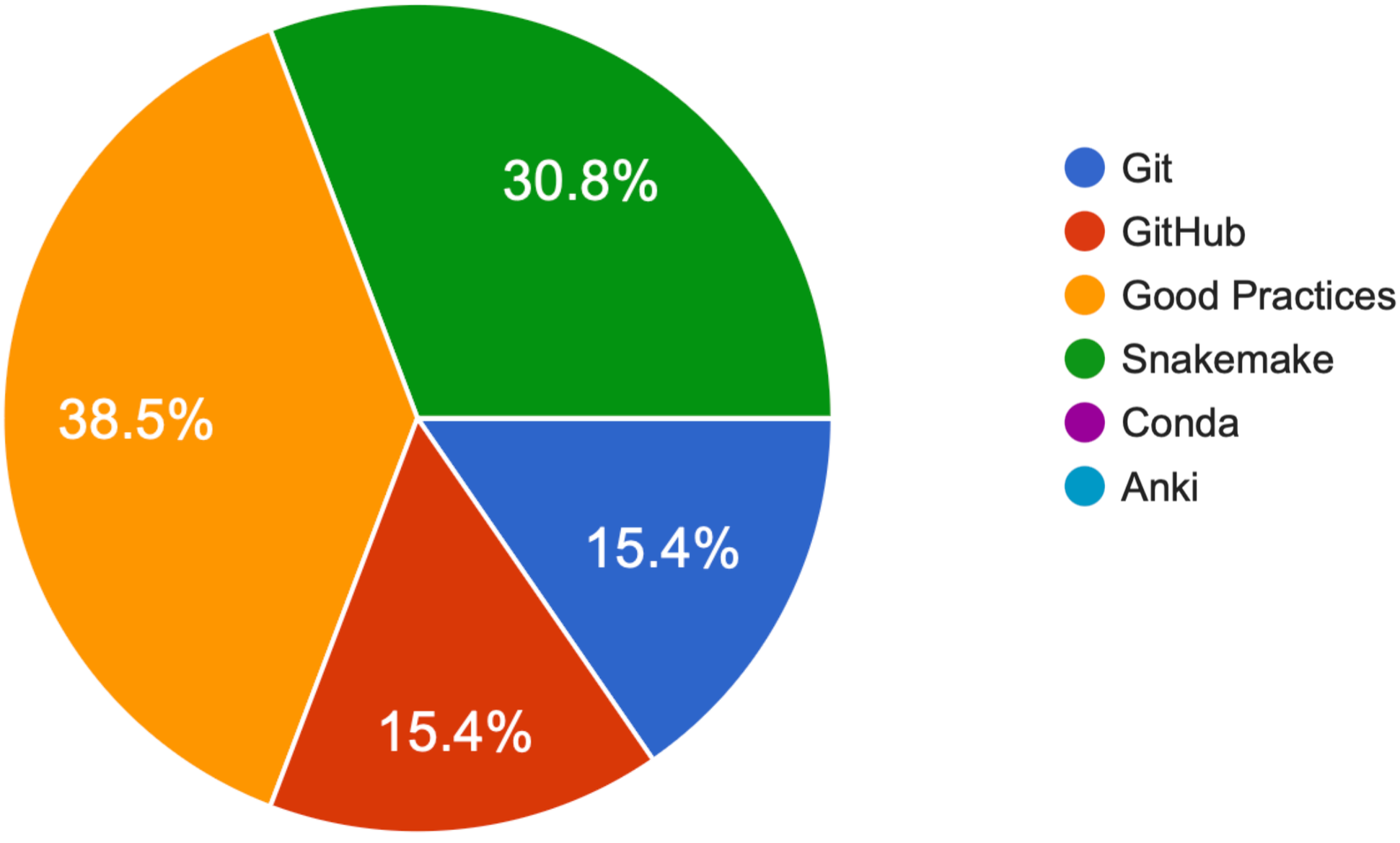
Next topic for Thursday the 28th

13 responses



Good practices

In Python

Pythonic

What’s pythonic?

What’s pythonic?

```
Python 3.10.8 | packaged by conda-forge | (main, Nov 22 2022, 08:27:35) [Clang 14.0.6 ]
In [2]: 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!
```

Readability counts

Non-Pythonic (Not Readable):

python Copy code

```python
def a(b,h):
    return b*h*0.5
```

Pythonic (Readable):

python Copy code

```python
def calculate_triangle_area(base: float, height: float) -> float:
    """
    Calculate the area of a triangle given its base and height.

    :param base: The length of the base of the triangle.
    :param height: The height of the triangle from its base.
    :return: Area of the triangle.
    """
    return 0.5 * base * height
```

PEP 8 style guide

- Indentation
- Maximum Line Length
- Imports
- Whitespaces
- Comments
- Naming Conventions
- Comparisons
- Documentation string

Styling example

python

Copy code

```python
import os  # Standard library import
import math  # Also a standard library import

# Constants (PEP 8 naming conventions)
MAX_RADIUS = 10

class Circle:  # CamelCase for classes
    """Represent a circle with a radius."""  # Docstring

    def __init__(self, radius: float) -> None:
        self.radius = radius

    def area(self) -> float:  # Function names in snake_case
        """Calculate the area of the circle."""  # Docstring
        return math.pi * self.radius * self.radius

def is_valid_radius(value: str) -> bool:
    """Check if the provided string can be a valid radius."""  # Docstring
    # Using `is not` for comparison
    if value is not None and value.isdigit():
        return True
    return False
```

Docstrings

python

```python
# Example function
def add(a, b):
    return a + b

# 1. PEP 257
def add_pep257(a, b):
    """Return the sum of two numbers."""
    return a + b

# 2. Google Style
def add_google(a, b):
    """
    Return the sum of two numbers.

    Args:
        a (int): The first number.
        b (int): The second number.

    Returns:
        int: The sum of a and b.
    """
    return a + b
```

```python
# 3. Numpydoc Style
def add_numpydoc(a, b):
    """
    Return the sum of two numbers.

    Parameters
    ----------
    a : int
        The first number.
    b : int
        The second number.

    Returns
    -------
    int
        The sum of a and b.
    """
    return a + b

# 4. Sphinx Style
def add_sphinx(a, b):
    """
    Return the sum of two numbers.

    :param a: The first number.
    :type a: int
    :param b: The second number.
    :type b: int
    :return: The sum of a and b.
    :rtype: int
    """
    return a + b
```

Numpy docs example

List comprehensions

Using Loops:

python Copy code

```python
numbers = [1, 2, 3, 4, 5]
even_squares = []

for n in numbers:
    if n % 2 == 0:
        even_squares.append(n**2)

print(even_squares)  # Output: [4, 16]
```

Using List Comprehensions:

python Copy code

```python
numbers = [1, 2, 3, 4, 5]
even_squares = [n**2 for n in numbers if n % 2 == 0]

print(even_squares)  # Output: [4, 16]
```

Vectorization

Using Loops:

python Copy code

```python
numbers = [1, 2, 3, 4, 5]
even_squares = []

for n in numbers:
    if n % 2 == 0:
        even_squares.append(n**2)

print(even_squares)  # Output: [4, 16]
```

Vectorization

python Copy code

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5])
even_squares = a[a % 2 == 0]**2

print(even_squares)  # Output: [4, 16]
```

Dictionary comprehensions

Using a Loop:

python Copy code

```python
data = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
filtered_data = {}

for key, value in data.items():
    if value % 2 != 0:
        filtered_data[key] = value

print(filtered_data)  # Output: {'a': 1, 'c': 3, 'e': 5}
```

Using Dictionary Comprehension:

python Copy code

```python
data = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
filtered_data = {key: value for key, value in data.items() if value % 2 != 0}

print(filtered_data)  # Output: {'a': 1, 'c': 3, 'e': 5}
```

Type hints

Without Type Hints:

python Copy code

```python
def merge_and_filter(dict1, dict2, threshold):
    merged_dict = {**dict1, **dict2}
    return {k: v for k, v in merged_dict.items() if v > threshold}
```

With Type Hints:

python Copy code

```python
from typing import Dict

def merge_and_filter(
    dict1: Dict[str, int],
    dict2: Dict[str, int],
    threshold: int
) -> Dict[str, int]:
    merged_dict = {**dict1, **dict2}
    return {k: v for k, v in merged_dict.items() if v > threshold}
```

Imports and Namespaces

python Copy code

```python
x = 10  # 'x' is in the global namespace.

def outer_func():
    y = 20  # 'y' is in the enclosing namespace of `outer_func`.

    def inner_func():
        z = 30  # 'z' is in the local namespace of `inner_func`.
        print(x, y, z)  # Accessible: x, y, z

    inner_func()  # Calling inner function
    # 'z' is out of scope here.

outer_func()  # Calling outer function

print(len)  # 'len' is in the built-in namespace.
```

Imports

python Copy code

```python
# 1. Importing an entire module
import math
print(math.sqrt(16))  # 4.0

# 2. Importing an entire module with an alias (a nickname)
import datetime as dt
print(dt.datetime.now())  # Prints the current date and time

# 3. Importing specific functions/classes from a module
from os import path
print(path.exists("myfile.txt"))  # Checks if "myfile.txt" exists

# 4. Importing multiple functions/classes from a module
from sys import argv, exit
print(argv)  # Prints the command-line arguments

# 5. Importing everything from a module (not recommended)
from random import *
print(randint(1, 10))  # Prints a random integer between 1 and 10
```

Exceptions

Function that throws an exception:

python Copy code

```python
def divide_numbers(numerator, denominator):
    if denominator == 0:
        raise ValueError("Denominator cannot be zero!")
    return numerator / denominator
```

Code that catches the exception:

python Copy code

```python
try:
    result = divide_numbers(10, 0)
    print(result)
except ValueError as e:
    print(f"Error occurred: {e}")
```

EAFP vs LBYL

python Copy code

```python
data = {"key1": "value1"}

# LBYL approach:
if "key1" in data:
    value = data["key1"]
else:
    value = "default_value"
print(value)  # Outputs: value1

# EAFP approach:
try:
    value = data["key1"]
except KeyError:
    value = "default_value"
print(value)  # Outputs: value1
```

Flat is better than nested

Bubble-Style:

```python
def process_data(data):
    if is_valid_type(data):
        if has_required_fields(data):
            if meets_other_criterion(data):
                # Actual processing logic here
                pass
            else:
                return "Data doesn't meet other criteria"
        else:
            return "Data lacks required fields"
    else:
        return "Data is of invalid type"
```

Gateway-Style:

```python
def process_data(data):
    if not is_valid_type(data):
        return "Data is of invalid type"

    if not has_required_fields(data):
        return "Data lacks required fields"

    if not meets_other_criterion(data):
        return "Data doesn't meet other criteria"

    # Actual processing logic here
    pass
```

Lambda functions

Without Lambda (Using a For Loop):

python Copy code

```python
filenames = ["data_01.txt", "temp_02.txt", "temp_03.log", "readme.md"]

temp_files = []

for name in filenames:
    if name.startswith("temp_"):
        temp_files.append(name)

print(temp_files)
```

Using a Lambda Function:

python Copy code

```python
filenames = ["data_01.txt", "temp_02.txt", "temp_03.log", "readme.md"]
temp_files = list(filter(lambda name: name.startswith("temp_"), filenames))
print(temp_files)
```

Iterators

Without Iterators (using a for-loop):

python Copy code

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []

for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)

print(even_numbers)  # Output: [2, 4, 6, 8, 10]
```

With Iterators (using Python's built-in `filter` function):

python Copy code

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4, 6, 8, 10]
```

Generators

Using Generators:

python

Copy code

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def even_numbers_gen(nums):
    for num in nums:
        if num % 2 == 0:
            yield num

even_numbers = even_numbers_gen(numbers)

print(list(even_numbers))  # Output: [2, 4, 6, 8, 10]
```

Use the `in` keyword

python

Copy code

```python
# Membership Testing with `in`
fruits = ["apple", "banana", "cherry"]
if "banana" in fruits:
    print("Yes, banana is in the fruits list.")

sentence = "Hello, World!"
if "World" in sentence:
    print("World is present in the sentence.")

person = {"name": "John", "age": 30}
if "name" in person:
    print("Name is a key in the dictionary.")

# Using `in` with a for loop
for fruit in fruits:
    print(fruit)
```

Use context managers

Without Context Manager:

python Copy code

```python
file = open("sample.txt", "r")
content = file.read()
print(content)
file.close()
```

With Context Manager:

python Copy code

```python
with open("sample.txt", "r") as file:
    content = file.read()
    print(content)
```

Use `enumerate()`

Without `enumerate`:

python Copy code

```python
colors = ['red', 'green', 'blue']

index = 0
for color in colors:
    print(f"Color at index {index} is {color}")
    index += 1
```

With `enumerate`:

python Copy code

```python
colors = ['red', 'green', 'blue']

for index, color in enumerate(colors):
    print(f"Color at index {index} is {color}")
```

Use keyword arguments

Without Keyword Arguments:

python Copy code

```python
def create_user(name, age, role):
    return f"Created user {name}, age {age}, as a {role}"

# Calling the function
user_info = create_user("Alice", 30, "Admin")
print(user_info)
```

With Keyword Arguments:

python Copy code

```python
def create_user(name, age, role):
    return f"Created user {name}, age {age}, as a {role}"

# Calling the function with changed order using keyword arguments
user_info = create_user(role="Admin", name="Alice", age=30)
print(user_info)
```

Use function argument defaults

Without Default Arguments:

python Copy code

```python
def connect_to_db(database, host):
    return f"Connecting to {database} on {host}"

connection = connect_to_db('my_database', 'localhost')
print(connection)  # "Connecting to my_database on localhost"
```

With Default Arguments:

python Copy code

```python
def connect_to_db(database, host='localhost'):
    return f"Connecting to {database} on {host}"

connection = connect_to_db('my_database')
print(connection)  # "Connecting to my_database on localhost"
```

Multiple assignment

python

Copy code

```python
# Basic multiple assignment
a, b, c = 1, 2, 3
print(a, b, c)  # Outputs: 1 2 3

# Swapping two variables
x, y = 5, 10
x, y = y, x
print(x, y)  # Outputs: 10 5

# Multiple assignment with a list
data = [7, 8, 9]
i, j, k = data
print(i, j, k)  # Outputs: 7 8 9

# Multiple assignment in a loop
pairs = [(1, 'one'), (2, 'two'), (3, 'three')]
for number, name in pairs:
    print(f"Number: {number}, Name: {name}")
```

Multiple assignment with functions

python

Copy code

```python
# Function returning multiple values
def get_name_and_age():
    return "Alice", 30

# Multiple assignment from function return
name, age = get_name_and_age()
print(f"{name} is {age} years old.")  # Outputs: Alice is 30 years old.
```

Packing/Unpacking positional arguments

python

Copy code

```python
# Using *args in function definition
def greet(*args):
    return f"Hello {args[0]}, you are {args[1]} years old."

greeting = greet('Alice', 30)
print(greeting)  # Outputs: Hello Alice, you are 30 years old.

# Unpacking a list or tuple as positional arguments
def welcome(name, age):
    return f"Hi {name}, you're {age} years old."

info_list = ['Bob', 25]
greeting_2 = welcome(*info_list)
print(greeting_2)  # Outputs: Hi Bob, you're 25 years old.
```

Packing/unpacking keyword arguments

python

Copy code

```python
# Using **kwargs in function definition
def greet(**kwargs):
    return f"Hello {kwargs['name']}, you are {kwargs['age']} years old."

greeting = greet(name='Alice', age=30)
print(greeting)  # Outputs: Hello Alice, you are 30 years old.

# Unpacking a dictionary as keyword arguments
def welcome(name, age):
    return f"Hi {name}, you're {age} years old."

info = {'name': 'Bob', 'age': 25}
greeting_2 = welcome(**info)
print(greeting_2)  # Outputs: Hi Bob, you're 25 years old.
```

Avoid using global variables

With Global Variables:

python Copy code

```python
counter = 0

def increment_counter():
    global counter
    counter += 1

increment_counter()
print(counter)  # Outputs: 1
```

Without Global Variables:

python Copy code

```python
def increment_counter(counter):
    return counter + 1

current_counter = 0
current_counter = increment_counter(current_counter)
print(current_counter)  # Outputs: 1
```

Don't compare explicitly to `True`, `False`, `None`

Not Recommended:

python

```python
# Explicit comparisons
value = True
if value == True:
    print("Value is True")

sequence = []
if sequence == []:
    print("Sequence is empty")

item = None
if item == None:
    print("Item is None")
```

Recommended:

python

```python
# Idiomatic comparisons
if value:
    print("Value is True")

if not sequence:
    print("Sequence is empty")

if item is None:
    print("Item is None")
```

Use Underscores for Multi-word Variable/Function Names:

python Copy code

```python
# Not Pythonic
userinput = ...
processdata(data)

# Pythonic
user_input = ...
process_data(data)
```

Chaining comparisons

Chaining Comparisons:

python Copy code

```python
# Not Pythonic
if x > 5 and x < 10:

# Pythonic
if 5 < x < 10:
```

Use `any()` and `all()`

Using `all()` and `any()`: For checking all or any items in a list against some condition.

python

Copy code

```python
# Check if all elements are True
all([item for item in my_list])

# Check if any elements are True
any([item for item in my_list])
```

Use f-strings

python

Copy code

```python
# Using f-string (Python 3.6+)
name = "Alice"
age = 30
message_fstring = f"My name is {name} and I am {age} years old."

# Using str.format() (Python 2.7+ and 3.0+)
message_format = "My name is {} and I am {} years old.".format(name, age)

# Using %-formatting (Python 2 and 3, but not recommended)
message_percent = "My name is %s and I am %d years old." % (name, age)

# Using string concatenation
message_concatenation = "My name is " + name + " and I am " + str(age) + "
```

Duck typing

python Copy code

```
def make_it_fly(bird):
    bird.fly()
```

python Copy code

```
class Duck:
    def fly(self):
        print("The duck is flying with wings!")
```

python Copy code

```
class Airplane:
    def fly(self):
        print("The airplane is flying with engines!")
```

python Copy code

```
d = Duck()
a = Airplane()

make_it_fly(d)  # Output: The duck is flying with wings!
make_it_fly(a)  # Output: The airplane is flying with engines!
```

Using `isinstance()`

python Copy code

```python
class Base: pass
class Derived(Base): pass

d = Derived()

isinstance(d, Base)     # True, because Derived is a subclass of Base
type(d) == Base         # False, because direct type of d is Derived
```

python Copy code

```python
x = 5
isinstance(x, (int, float, str))  # True, checks if x is any of int, float,
```

Object-oriented programming

Procedural Approach

python

```python
def create_account(initial_balance):
    return {"balance": initial_balance}

def deposit(account, amount):
    account["balance"] += amount
    return account

def withdraw(account, amount):
    if amount > account["balance"]:
        print("Insufficient funds!")
        return account
    account["balance"] -= amount
    return account

def get_balance(account):
    return account["balance"]

# Usage
account = create_account(100)
deposit(account, 50)
withdraw(account, 30)
print(get_balance(account))  # Output: 120
```

Object-Oriented Approach

python

```python
class BankAccount:
    def __init__(self, initial_balance):
        self.balance = initial_balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds!")
            return
        self.balance -= amount

    def get_balance(self):
        return self.balance

# Usage
account = BankAccount(100)
account.deposit(50)
account.withdraw(30)
print(account.get_balance())  # Output: 120
```

Reuse code

Without Code Reuse:

python Copy code

```python
def calculate_total_without_discount(items):
    total = 0
    for item in items:
        total += item['price']
    return total

def calculate_total_with_discount(items, discount):
    total = 0
    for item in items:
        total += item['price'] - (item['price'] * discount / 100)
    return total
```

With Code Reuse:

python Copy code

```python
def calculate_total(items, discount=0):
    total = 0
    for item in items:
        total += item['price'] * (1 - discount / 100)
    return total
```

Familiarize with popular libraries

- **Pandas**: A data manipulation library offering data structures for efficiently storing large datasets and tools for reshaping, aggregating, and filtering data.
- **NumPy**: A foundational package for numerical computations, providing support for large multi-dimensional arrays and matrices.
- **SciPy**: An open-source library used for high-level computations in mathematics, science, and engineering, built on the NumPy extension.
- **Matplotlib/Seaborn**: Matplotlib is a comprehensive plotting library for static, animated, and interactive visualizations, while Seaborn is a statistical data visualization library built on top of Matplotlib.
- **BioPython**: A collection of tools and libraries for computational biology, bioinformatics, and related fields.

That’s it