# Python: Day 04

Advanced Programming

# Agenda

01

## Packaging

Handling Python Files

02

## Multiple Tasks

Handling bottlenecks

03

## Best Practices

Writing Pythonic code

04

## Testing

Code correctness

05

## Web Dev

Introduction to Flask

06

## Lab Session

Culminating Exercise

01

# Packaging

How to handle Python files properly

# Modules and Packages

## Module

Single Python file

```
.
└── module.py
```

## Package

Folder with an `__init__.py`

```
.
└── package/
    ├── __init__.py
    └── module.py
```

# Basic Import

./hello.py

```python
def say_hello():
    print("Hello from module hello")
```

./current_file.py

```python
import hello

hello.say_hello()
```

# Specific Import

./hello.py

```python
def say_hello():
    print("Hello from module hello")
```

./current_file.py

```python
from hello import say_hello

say_hello()
```

# Basic Import with Alias

./hello.py

```python
def say_hello():
    print("Hello from module hello")
```

./current_file.py

```python
import hello as ho

ho.say_hello()
```

# Multiple Specific Import

./hello.py

```
def say_hello():
    print("Hello from module hello")
greeting = “Yellow!”
```

./current_file.py

```
from hello import say_hello, greeting

say_hello()
print(greeting)
```

# Basic Nested Import

./package/module_01.py

```python
def say_hello():
    print("Hello from module 1!")
```

./current_file.py

```python
import package.module_01

package.module_01.say_hello()
```

# Specific Nested Import

./package/module_01.py

```python
def say_hello():
    print("Hello from module 1!")
```

./current_file.py

```python
from package.module_01 import say_hello

say_hello()
```

# Nested Import with Alias

./package/module_01.py

```python
def say_hello():
    print("Hello from module 1!")
```

./current_file.py

```python
import package.module_01 as pm1

pm1.say_hello()
```

# Standard Packaging Format

Most Python projects follow this project structure:

```
project_name/
    ├── LICENSE
    ├── pyproject.toml
    ├── README.md
    ├── src/
    │   ├── example_package_1/
    │   │   ├── __init__.py
    │   │   └── example.py
    │   └── example_package_2/
    │       ├── __init__.py
    │       └── example.py
    ├── tests/
    ├── doc/
    └── script/
```

# Try these Built-in Libraries!

### Math

Common math constants and operations

### Datetime

Dedicated package for handling calendar dates

### Collections

Additional data structures

### Time

Access to system time, delays, and conversions

### SQlite

Quick setup for a light database system

### Itertools

Efficient looping and combinatorials

H1

# Random Counter

Using pre-built packages to do our work

# Random Counter

Create one million random numbers from one to one thousand.

```
random_numbers = [...]
```

List down the number of occurrence for each number

```
random_number_count = ...
```

Finally, print out the number with the highest count and how many times it appeared

02

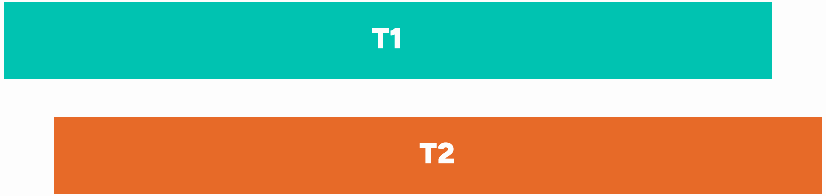
# Multiple Tasks

A preview of Multiprocessing and Multithreading

# Parallelism versus Concurrency

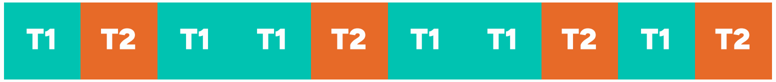
## Parallel Process

Tasks running simultaneously
or at the same time



## Concurrent Process

Switching between tasks
when waiting for results



# Concurrency

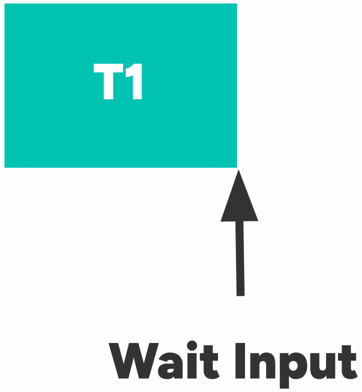
Working while waiting for other tasks

# Concurrent Process

### Current Task

T1

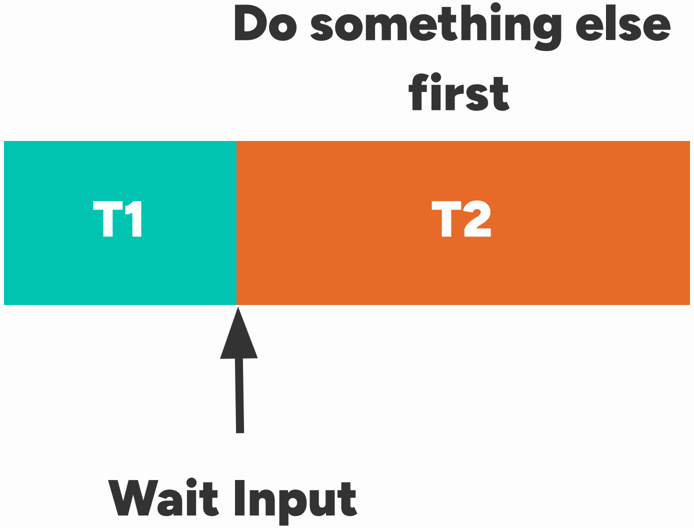
# Concurrent Process



T1

Wait Input

# Concurrent Process



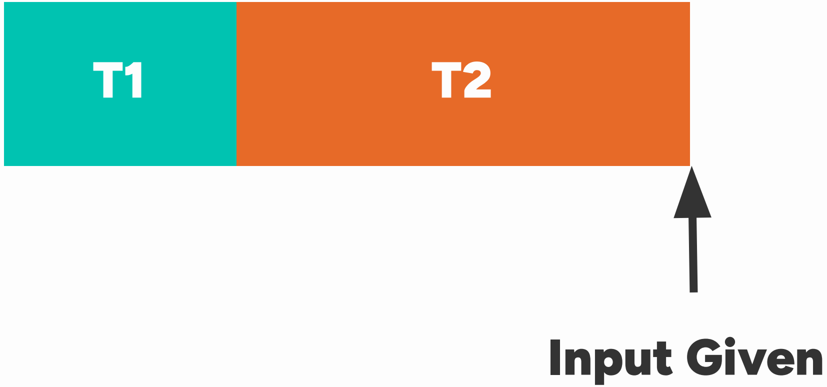
Do something else first

T1

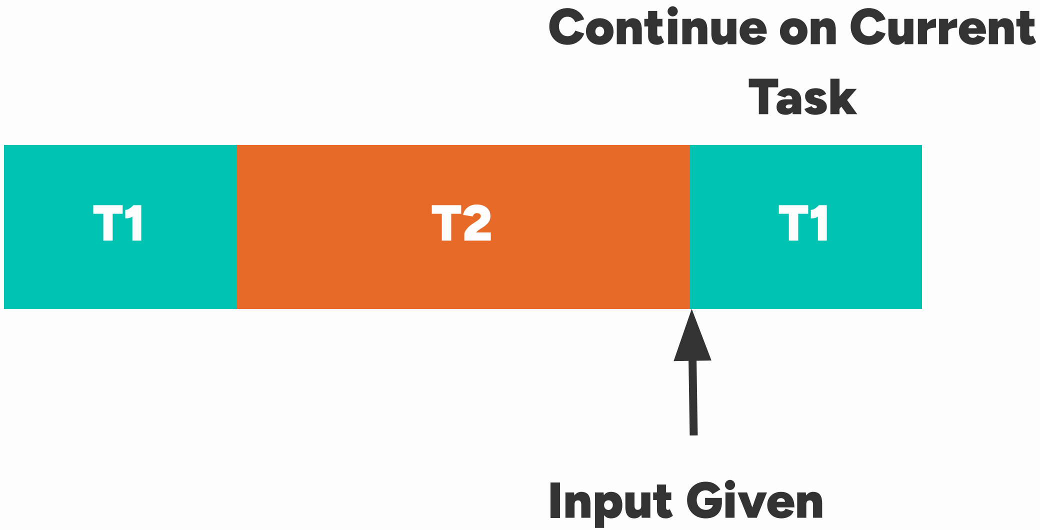
T2

Wait Input

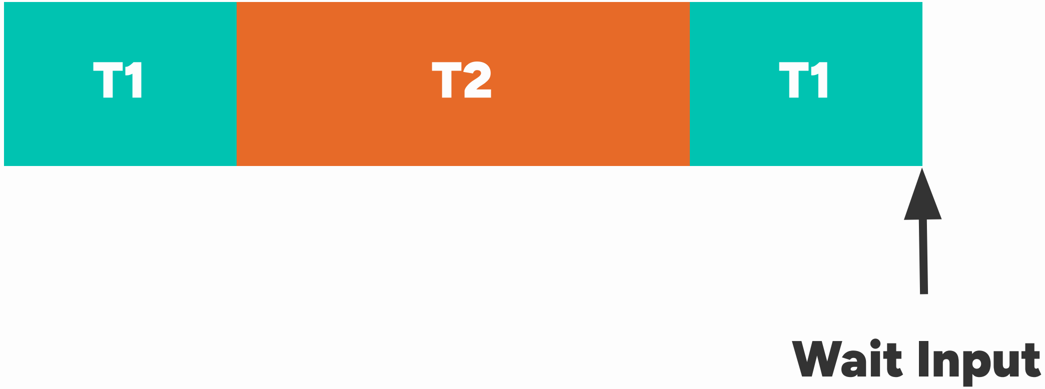
# Concurrent Process



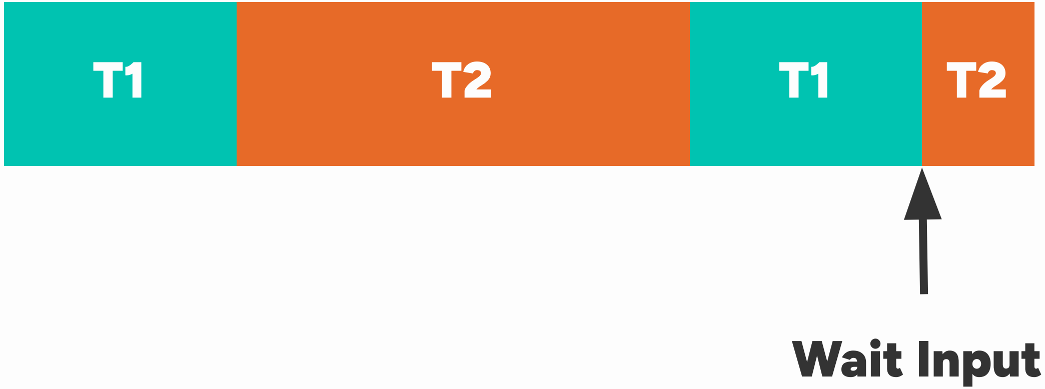
# Concurrent Process



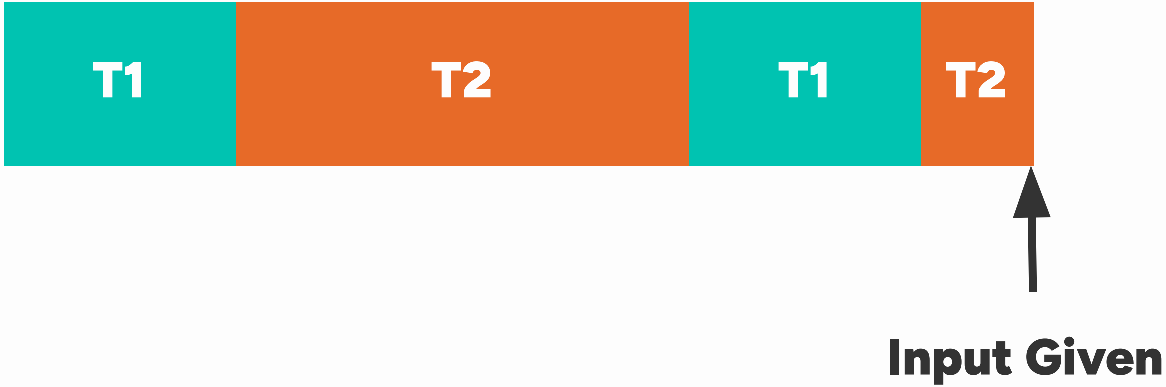
# Concurrent Process



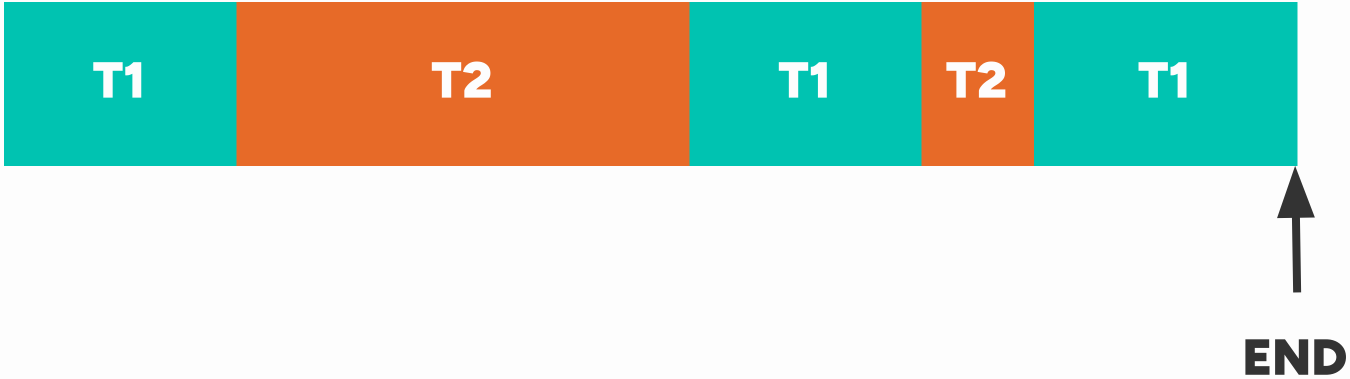
# Concurrent Process



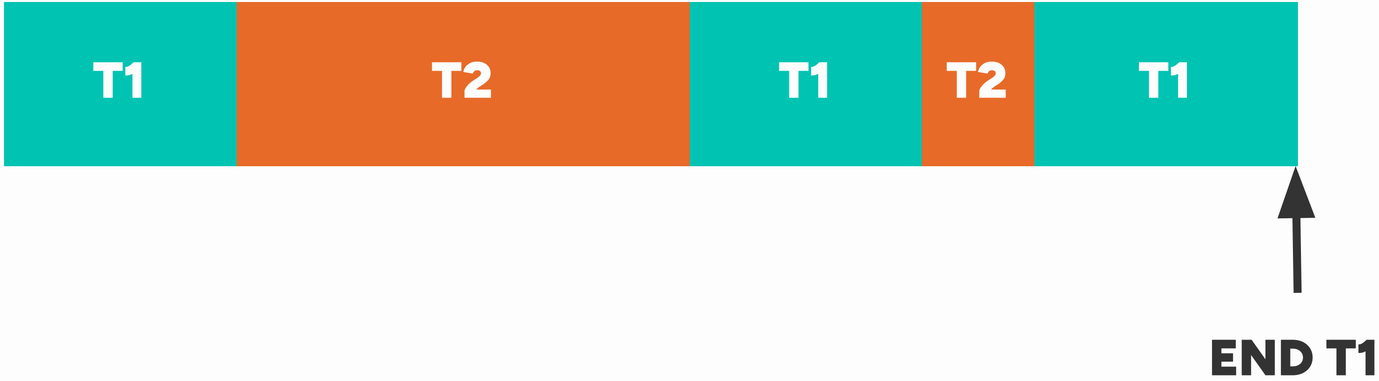
# Concurrent Process



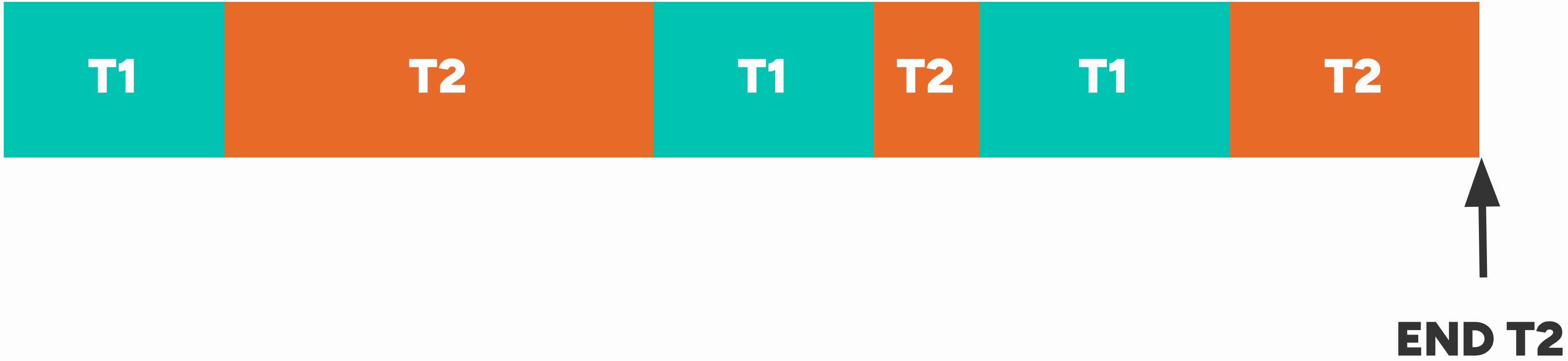
# Concurrent Process



# Concurrent Process

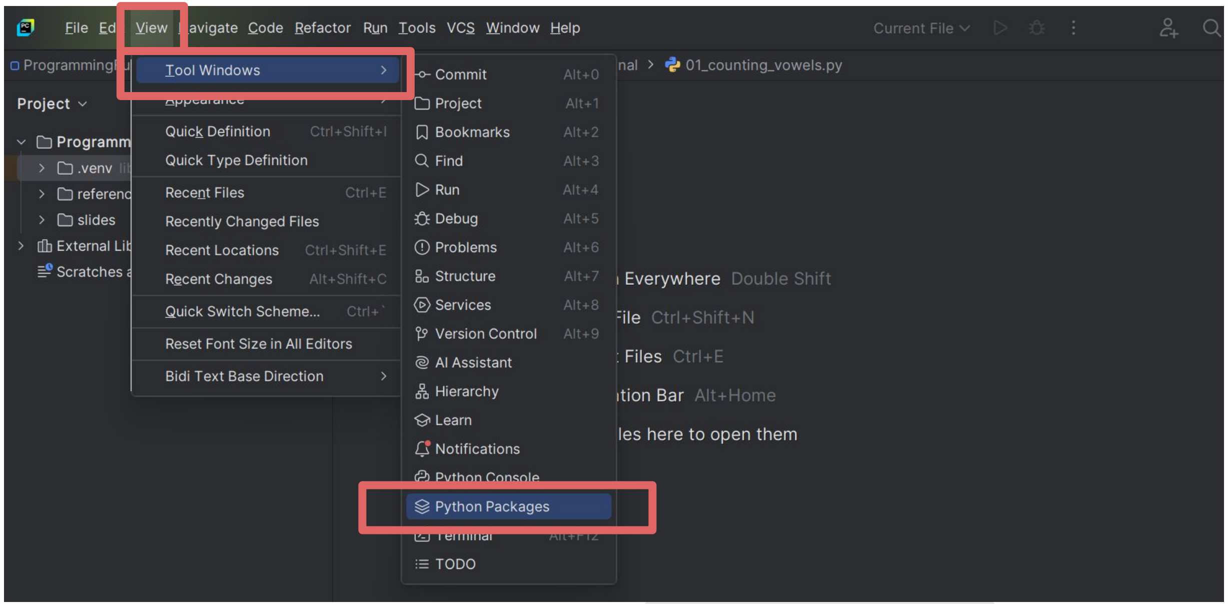


# Concurrent Process



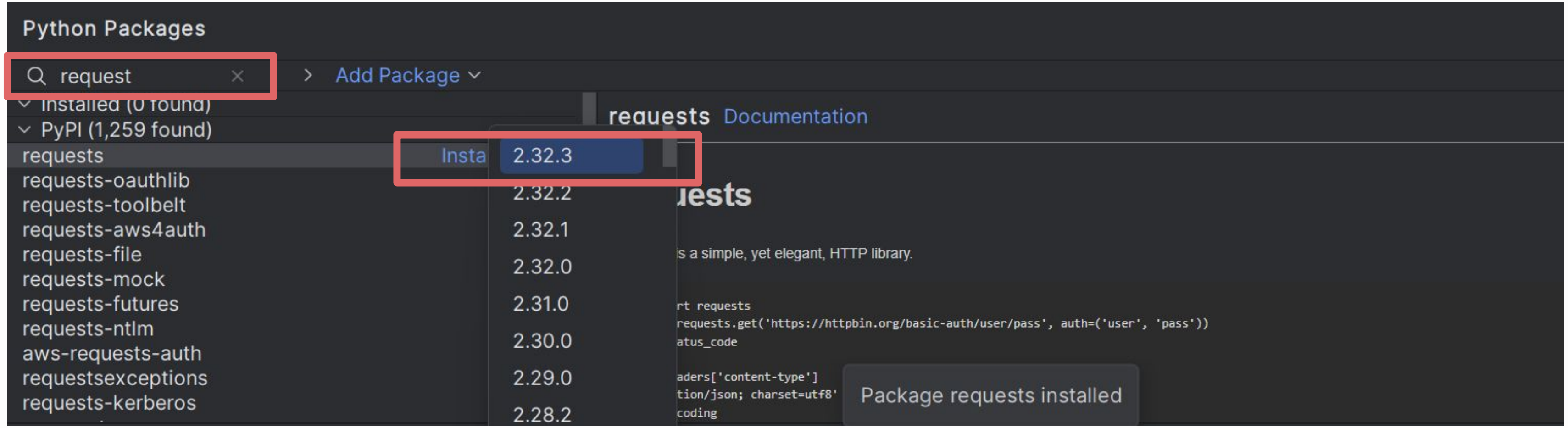
# Prerequisite: Python Packages

In the upper left menu navigation bar select **`View > Tool Windows > Python Packages`**



# Prerequisite: Download Request Packages

A new menu will open on the lower right. Search for the **`request`** library. Then select **`install`**. Make sure to select the latest version available.



is a simple, yet elegant, HTTP library.

```
rt requests
requests.get('https://httpbin.org/basic-auth/user/pass', auth=('user', 'pass'))
atus_code

aders['content-type']
tion/json; charset=utf8'
coding
```

Package requests installed

# Thread Pool Submission

```
import concurrent.futures
import time

def process(number):
    _ = number * 1_000_000 ** 1_000_000
    print(“Finished computation”)

if __name__==”__main__”:
    start_time = time.time()
    with concurrent.futures.ThreadPoolExecutor() as executor:
        x = executor.submit(process, 3)
        y = executor.submit(input, “Enter number: ”)

    end_time = time.time()
    print(end_time - start_time)

```

# Thread Pool Mapping

```python
import concurrent.futures
import requests
import time

def fetch_url(url):
    return requests.get(url).status_code

urls = [ 'https://httpbin.org/delay/5',
         'https://httpbin.org/delay/7']
if __name__==”__main__”:
    start_time = time.time()
    with concurrent.futures.ThreadPoolExecutor() as executor:
        results = executor.map(fetch_url, urls)

    end_time = time.time()
    print(end_time - start_time)
```

H2

# Website Check

Check multiple websites if they are working

# Website Check

```python
import concurrent.futures
import requests
import time

def check_website(url):
    try:
        response = requests.get(url)
        if response.status_code == 200:
            print(f"{url} is up!")
        else:
            print(f"{url} status {response.status_code}")
    except:
        print(f"{url} failed to reach.")

```

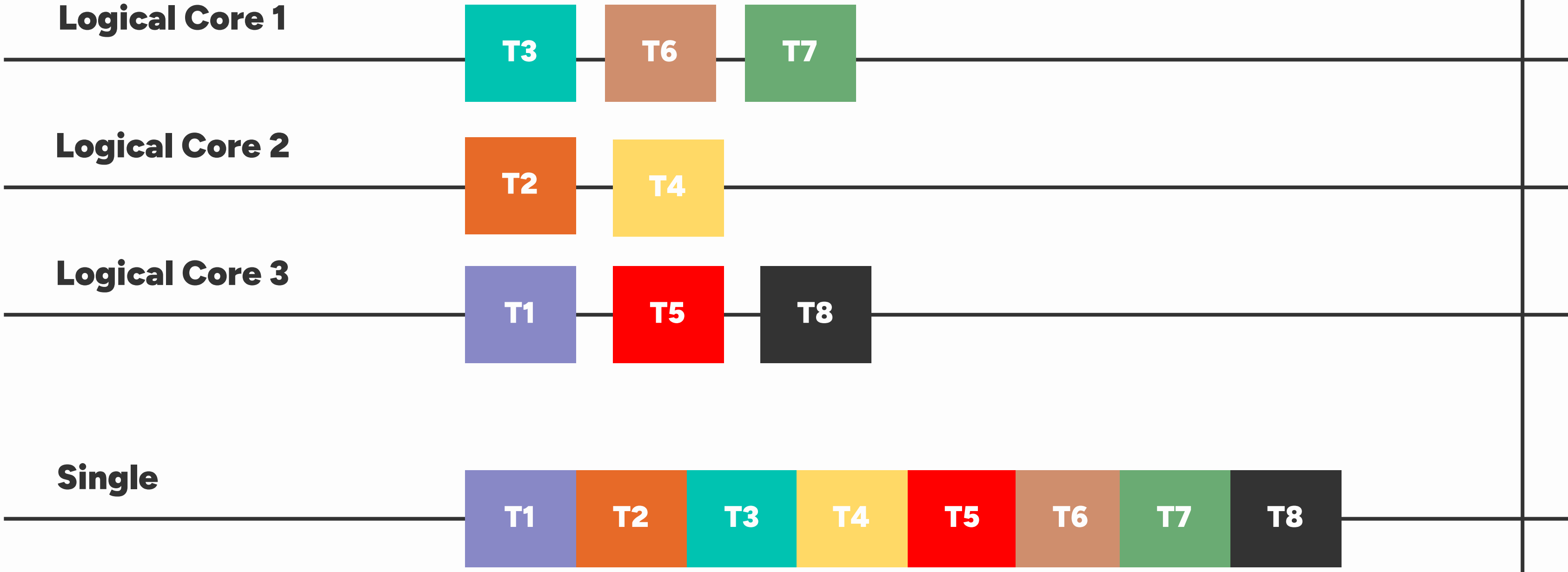
# Manual Task

```python
def read_websites(file_path):
    with open(file_path, 'r') as file:
        websites = file.readlines()
        return [website.strip() for website in websites]

start_time = time.time()

websites = read_websites('websites.txt')
with concurrent.futures.ThreadPoolExecutor() as executor:
    executor.map(check_website, websites)

end_time = time.time()
print(end_time - start_time)
```

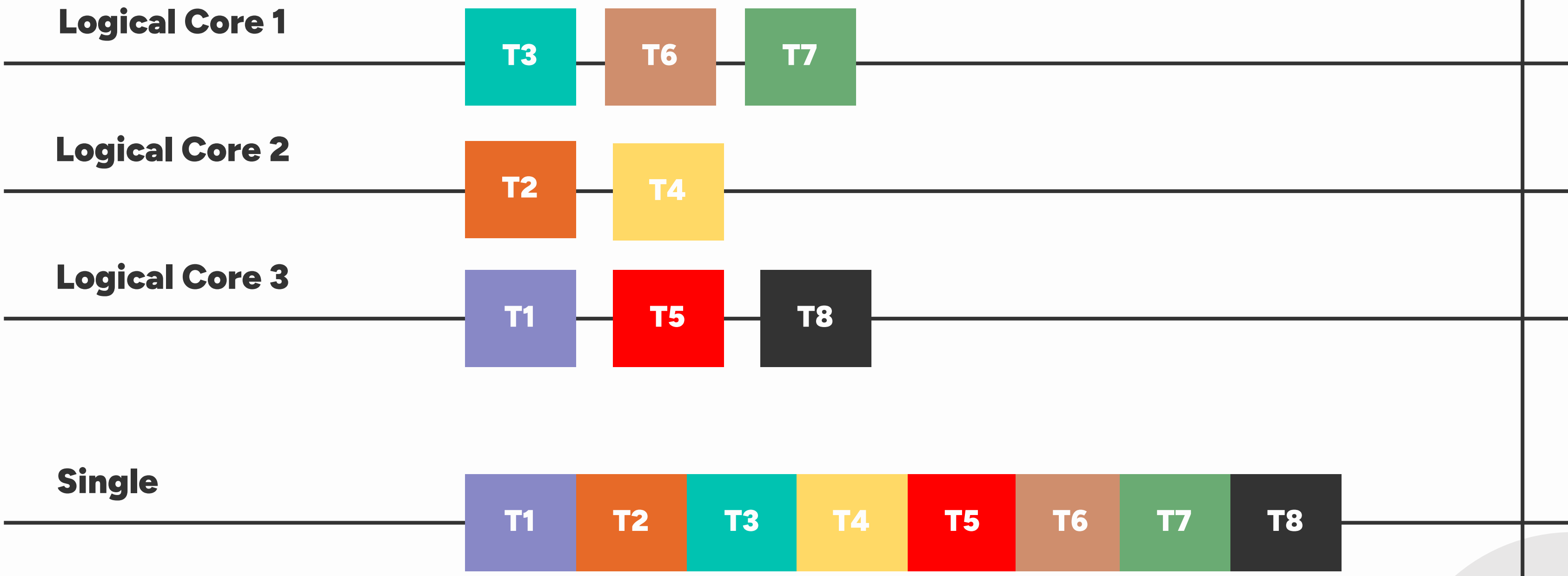
# Multiprocessing

Actually doing multiple tasks at once

# Parallelism using Multiprocessing



# Parallelism using Multiprocessing



# Sequential Task

```
import multiprocessing
import time

def process(number):
    return number * 1_000_000 ** 1_000_000

if __name__==”__main__”:
    start_time = time.time()

    numbers = [(number + 1) for number in range(3)]
    results = [process(number) for number in numbers]

    end_time = time.time()
    print(end_time - start_time)

```

# Multi-Process Task

```
from multiprocessing import Pool
import time

def process(number):
    return number * 1_000_000 ** 1_000_000

if __name__==”__main__”:
    start_time = time.time()

    numbers = [(number + 1) for number in range(3)]
    with Pool() as pool:
        results = pool.map(process, numbers)

    end_time = time.time()
    print(end_time - start_time)
```

H3

# Fibonacci Task

Fancy counting done fast

# Sequential Fibonacci Calculation

```python
from multiprocessing import Pool
import time

def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

if __name__==”__main__”:
    start_time = time.time()
    numbers = [35, 36, 37, 38]
    for number in numbers:
        print(f"Fibonacci({number}) = {fibonacci(number)}")

    end_time = time.time()
    print(end_time - start_time)
```

03

# Best Practices

Recommended way to write Python code

# Example Code No. 1

```python
def function(s):
    ws = s.split()

    vc = 0
    vs  = "aeiou"

    for w in ws:
        if any(v in w for v in vs):
            vc += 1

    return vc
```

# Example Code No. 1 (Refactor)

```python
def count_words_with_vowel(text):
    words = text.split()

    words_with_vowels_count = 0
    vowels= "aeiou"

    for word in words:
        if any(vowel in words for vowel in vowels):
            words_with_vowels_count += 1

    return words_with_vowels_count
```

# Example Code No. 2

```python
def function(is):
    ic = {}

    for i in is:

        if i in ic:
            ic[i] += 1
        else:
            ic[i] = 1

    return ic
```

# Example Code 2 (Refactor)

```python
def count_per_item(items):
    item_count = {}

    for item in items:

        if item in item_count:
            item_count[item] += 1
        else:
            item_count[item] = 1

    return item_count
```

“Code is read much more often than it is written.”

**— Guido van Rossum**

# import this

If the implementation is **hard to explain** , it's a **bad idea**

# Programming Principles

## Don’t Repeat Yourself

Code duplication is a sign to use variables, functions, classes, and loops

## Keep it Simple, Silly

Always aim for the simplest approach to the code

## Loose Coupling

Minimize dependency of functions and classes with each other

## Abstraction

Hide details in classes and functions to make things simpler at a quick glance

# Python Enhancement Proposal (PEP) 8

## Consistency

Makes it easier to read code quickly out of experience

## Maintenance

PEP 8 is built for the purpose of making code easier to debug

## Community

PEP 8 reflects the format and conventions that communities use

# PEP 8 Quick Notes

### Use 4 Spaces

Don’t use tabs and especially don’t mix spaces and tab

### Limit to 79 Chars

Limit lines (72 characters for comments) to make code more readable or digestible

### Start Private

If you’re not sure, start private as it’s harder to go from public to private

### Naming Convention

Use snake_case for variables, functions, and files. Use PascalCase for classes.

# PEP 8 Long Statements

For long operations, place the operator at the front

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

```
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

# PEP 8 Extra Whitespaces

Avoid extra spaces as it is unnecessary

```
spam(ham[1], {eggs: 2})
```

```
spam( ham[ 1 ], { eggs: 2 } )
```

```
dct['key'] = lst[index]
```

```
dct ['key'] = lst [index]
```

```
x             = 1
y             = 2
long_variable = 3
```

# PEP 8 Implicit Boolean Checks

If your variable is a Boolean, don’t use an equality check (remember, it auto-uses `bool()` )

```
if greeting == True:
```

```
if greeting is True:
```

```
if greeting:
```

# Documentation

## Provide Some Context

Note all of the prerequisites or key insights needed to understand a process. Mainly, explain why you are doing it

## Enhance Readability

If a process is really hard to understand, explain it in alternative ways of phrasing

## Summarize Immediately

One line can summarize paragraphs or entire documents depending on the use case

# Hallmarks of a Good Comment

### Clear

Very specific and relevant

### Updated

Outdated code is a severe liability

### Not Redundant

Provide information not yet revealed

### Proper Grammar

Keep it professional

### Simple

A New Developer should follow it

### References

Provide links to related or source of truth

# Inline Comments

Inline comments can be used to make quick notes or one-off explanations on why

```
# Convert temperature from Celsius to Fahrenheit
temperature_f = (temperature_c * 9/5) + 32
```

```
# This is a variable
x = 10

# This prints x
print(x)
```

# Docstrings

Docstrings are commonly used to document functions (summary, args, return, errors).

```python
def calculate_circle_area(radius):
    """
    Return the area of a circle with the given radius.

    Args:
        radius (float): Circle's radius. Must be non-negative.

    Returns:
        float: Area of the circle.

    Raises:
        ValueError: If radius is negative.
    """
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * radius ** 2
```

# Docstrings

Docstrings can still be used for simple functions. In this case, they span for a single line

```
def greet():
    """Print a simple greeting message."""
    print("Hello, welcome!")
```

# Docstrings

Besides the documentation on-hover, you can use docstrings to provide support for `help`

```
help(calculate_circle_area)
```

# Docstrings

Docstrings can also be used for classes.

```
class VideoPlayer:
     """Provides convenient functions for playing and processing video files”""


     def __init__(self, video):
          """Provides convenient functions for playing and processing video files

          Args:
               video (str): Filename of video

          """
          self.video = video
```

# Variable Naming

Yes, it needs its own section

# Consistent Variable Names

Do not suddenly shift your themes or word choice in-between cod

```
customer_name = "John Doe"
client_age = 30 customer
shopper_order = ["apple", "banana", "orange"]
```

```
customer_name = "John Doe"
customer_age = 30 customer
customer_order = ["apple", "banana", "orange"]
```

# Avoid Abbreviations

It seems to make sense when you made it. But will we remember after a few weeks?

```
hrb = 5000
```

# Avoid Abbreviations

Make it very clear from the get-go

```
hrb = 5000
```

```
human_resources_budget = 5000
```

# Descriptive Variables

The variable name should be enough

```
x = 10
y = [1, 2]
data = “yes”
```

```
total_items = 10
list_of_attendees_per_day = [1, 2]
question01_response = “yes”
```

# Type Hinting

Saving yourself future debugging headaches

# Type Hinting (Input)

You can provide a hint on what data type you’re expecting for function parameters

```
def add(number1: int, number2: int):
    “””Returns the mathematical summation of the two numbers.

    Args:
        number1 (int): First addend in summation
        number2 (int): Second addend in summation

    Returns:
        int: Addition of the two numbers
    “””
    return number1 + number2
```

# Type Hinting (Output)

You can provide a hint on what data type you’re expecting for function outputs

```
def add(number1: int, number2: int) -> int:
    “””Returns the mathematical summation of the two numbers.

    Args:
        number1 (int): First addend in summation
        number2 (int): Second addend in summation

    Returns:
        int: Addition of the two numbers
    “””
    return number1 + number2
```

# Type Hinting (Complete)

You can support more than one type of hinting

```
def add(number1: int|float, number2: int|float) -> int|float:
    “””Returns the mathematical summation of the two numbers.

    Args:
        number1 (int|float): First addend in summation
        number2 (int|float): Second addend in summation

    Returns:
        int|float: Addition of the two numbers
    “””
    return number1 + number2
```

# Type Hinting Examples

There are a lot of built-in type hints for the standard data types and for nested data types

```
variable1: int = 1
variable2: list[int] = [1, 2, 3]
variable3: dict[str, int] = {“a”: 123, “b”: 456, “c”: 890}
variable4: dict[str, list[int]] = {“num1”: [1, 2, 3], “num2”: [4]}
variable5: tuple[int, int] = (0, 1)
variable6: list[tuple[int, int]] = [(9, 1), (2, 3), (5, 2)]
```

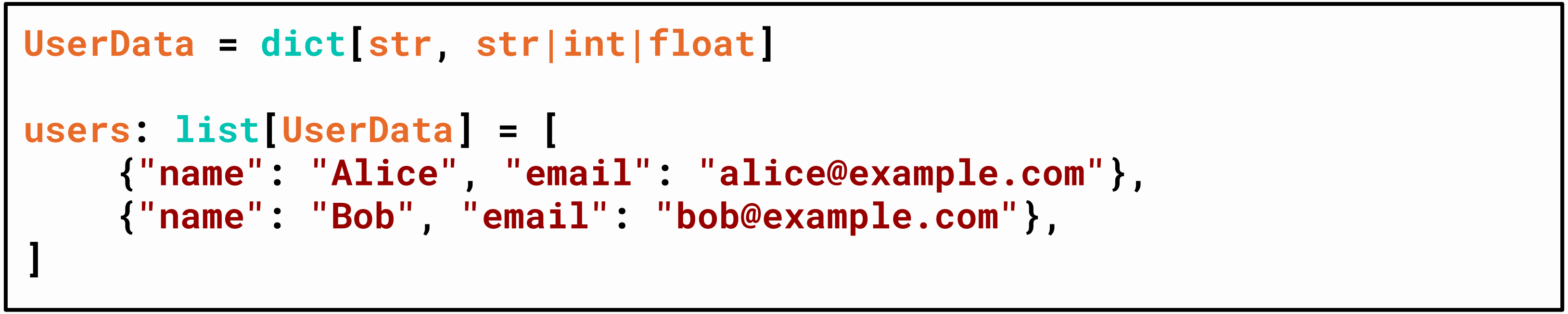
# Variable Type Hinting

Type hints also work for regular variables. Here is an example of the syntax for data structures

```
total_tasks: int = 81

points: list[int] = [1, 2, 3]
priority: tuple[str, str, str] = [“low”, “medium”, “urgent”]

employees: dict[int, str] = dict()
employees.update({9823: “Jay”, 1821: “Caroline”})

downtime_logs: list[ dict[str, str] ] = [
    {“Engineering”: “Lunch”, “Finance”: “Team Building”},
    {“Security”: “Maintenance”},
    {“Hiring”: “Tax Filing”, “Engineering”: “System Update”},
]
```

# Complex Type Hinting

For type hinting that is hard to read due to nesting, type hints can be stored in variables



```python
UserData = dict[str, str|int|float]

users: list[UserData] = [
    {"name": "Alice", "email": "alice@example.com"},
    {"name": "Bob", "email": "bob@example.com"},
]
```

# Typing Module

The typing module has additional typing and syntax for convenience

```
from typing import Literal, Iterable

priority = Literal[“low”, “medium”, “urgent”]
priorities: list[priority] = [“medium”, “urgent”, “urgent”, “low”]

def urgent_points(items: Iterable) -> int:
    urgent_point: int= 10
    return sum(urgent_point for item in items if item == “urgent”)
```

H4

# Code Review

Let’s assess how to improve code

# Improve this code:

```
def u(p):
    v = 1
    for w in range(1, p + 1):
    v *= w
    return v

x = 5
y = u(x)
print(y)
```

# Improve this code:

```
def m(n):
    p = True
    for q in range(2, n):
    if n % q == 0:
        p = False
        break
    return p

r = 29
s = m(r)
print(s)
```

# Improve this code:

```
def m(n):
    p = []
    for q in n:
    if q not in p:
        p.append(q)
    return p

r = [1, 2, 3, 3, 4, 5, 5]
s = m(r)
print(s)
```

# Testing

Security for your colleagues and future self

# Common Types of Testing

## Unit

Testing individual parts or functions in isolation

## Integration

Testing if different components work together correctly

## Regression

Testing if changes in the code doesn’t accidentally break anything

# Pytest Framework

The **pytest** framework is one of the most common testing frameworks, known for its simplicity, scalability, and powerful features.

```
$ pip install pytest
```

For as long as the function has **test** at the start of its name, it will be detected as a test.

```python
def test_sanity():
    assert len([99, 98, 97]) == 3
```

```
$ pytest
```

# Unit Test

Testing individual components or functions in isolation from other parts

```python
def square(x):
    return x * x

def test_square():
    assert square(2) == 4
    assert square(-3) == 9
    assert square(0) == 0
    print("All unit tests passed!")

test_square()
```

# Integration Test

Testing if different components work as intended when combined together

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

def square(x):
    return x * x

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

```

# Integration Test

Testing if different components work as intended when combined together

```
def calculate_expression(x, y):
    return add(square(x), multiply(y, 2))

def test_calculate_expression():
    assert calculate_expression(2, 3) == 10
    assert calculate_expression(0, 5) == 10

    print("All integration tests passed!")

test_calculate_expression()
```

# Regression Test

Check if changes in the code have not affected existing functionality

```python
def calculate_expression(x, y, z=0):
    return add(square(x), multiply(y, 2)) - z

def test_calculate_expression():
    assert calculate_expression(2, 3) == 10
    assert calculate_expression(0, 5) == 10
    assert calculate_expression(2, 3, 2) == 10
    print("All integration tests passed!")

test_calculate_expression()
```

# Pytest Classes

Tests can be grouped into classes for further organization

```python
class TestClass:
    def test_one(self):
        word = "this"
        assert "h" in word

    def test_two(self):
        word = "hello"
        assert hasattr(word, "check")
```

# Standard Packaging Format (Review)

Most Python projects follow this project structure:

```
.
└── project_name/
    ├── ...
    └── src/
        ├── example_package_1/
        ├── example_package_2/
        └── tests
            ├── example_package_1/
            │   └── test_package_1.py
            └── example_package_2/
                └── test_package_.py
```

H5

# Intentional Bug

A surprising amount of time is invested here

# Fix the possible bug

```
def find_even_numbers(numbers):
    evens = []
    for num in numbers:
        if num % 2 == 1:
            evens.append(num)
        return evens

numbers = [1, 2, 3, 4, 5, 6]
print(f"Even numbers: {find_even_numbers(numbers)}")
```

# Fix the possible bug

```
def remove_duplicates(numbers):
    for num in numbers:
        if numbers.count(num) > 1:
            numbers.remove(num)
        return numbers

numbers = [1, 2, 2, 3, 3, 4]
print(f"Unique numbers: {remove_duplicates(numbers)}")
```

# Fix the possible bug

```
def average(numbers):
    total = 0
    for num in numbers:
        total += num
    return total / len(nums)

numbers = [10, 20, 30, 40]
print(average(numbers))
```

# Fix the possible bug

```
def count_positive_numbers(numbers):
    count = 0
    for num in numbers:
    if num > 0:
        count += 1
    else:
        count -= 1
    return count

numbers = [1, -2, 3, 4, -5, 6]
print(count_positive_numbers(numbers))
```

04

# Web Dev

Providing online access to your business logic

# Web Frameworks

## Flask

- Minimalist and lightweight
- Freedom to choose tools for each part
- **Small and Fast Web Applications**

## Django

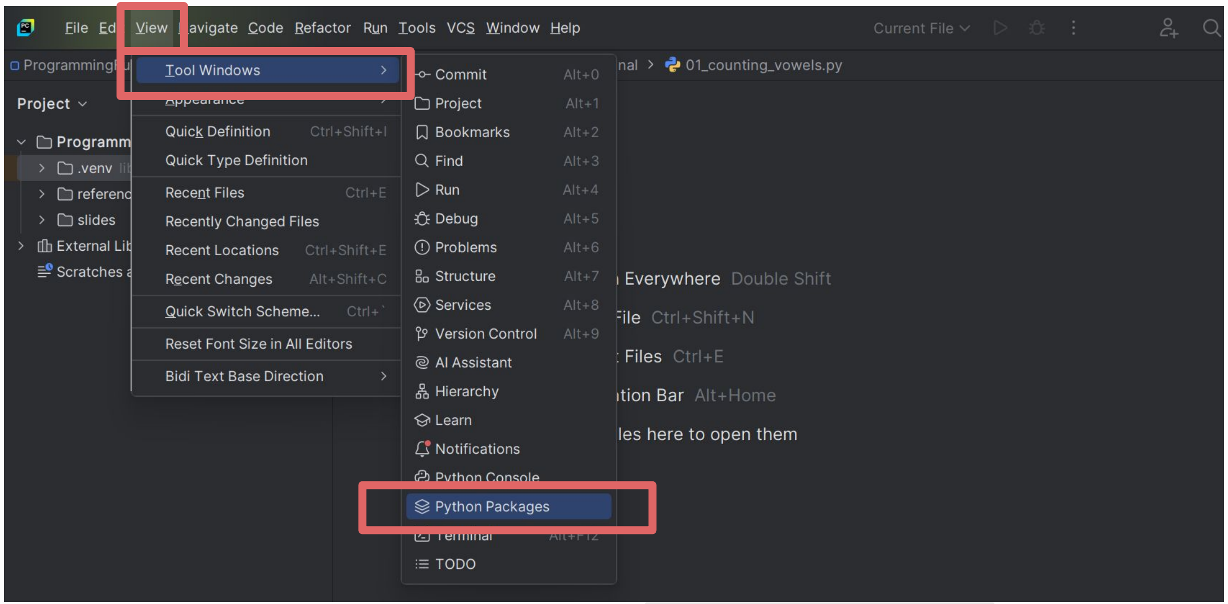
- Multiple out-of-the-box features
    - Object Relational Mapping
    - Fully functional Admin Panel
    - Security Measures and Authentication
- **Medium to Large Web applications**

# Initial Setup

Package download and Initial Page

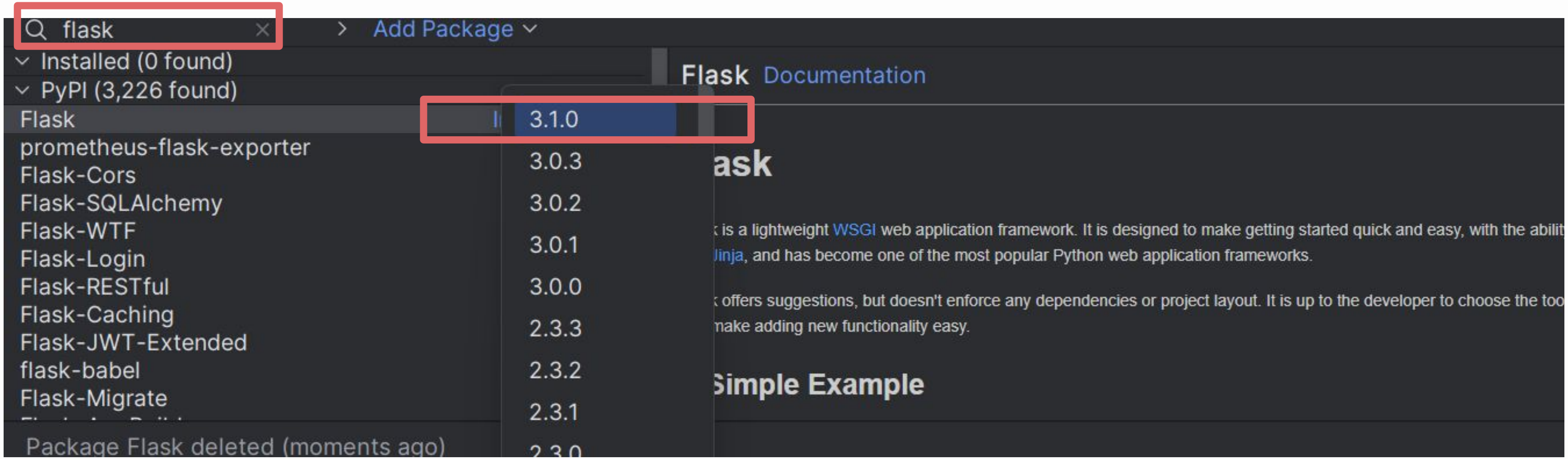
# Prerequisite: Python Packages

In the upper left menu navigation bar select **`View > Tool Windows > Python Packages`**



# Prerequisite: Download Request Packages

A new menu will open on the lower right. Search for the **`flask`** library. Then select **`install`**. Make sure to select the latest version available.



Package Flask deleted (moments ago)

k is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the abilit Jinja, and has become one of the most popular Python web application frameworks.

k offers suggestions, but doesn't enforce any dependencies or project layout. It is up to the developer to choose the too make adding new functionality easy.

# Minimum Setup

```
from flask import Flask

app = Flask(__name__)
app.run()
```

# Routing

Setting up the subpages of the site

# Index Route

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "Index Page"

app.run()
```

# Additional Route

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "Index Page"

@app.route("/profile/")
def profile():
    return "Profile Page"

app.run()


```

# Route Aliasing

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "Index Page"

@app.route("/profile/")
@app.route("/profiles/")
def profile():
    return "Profile Page"

app.run()

```

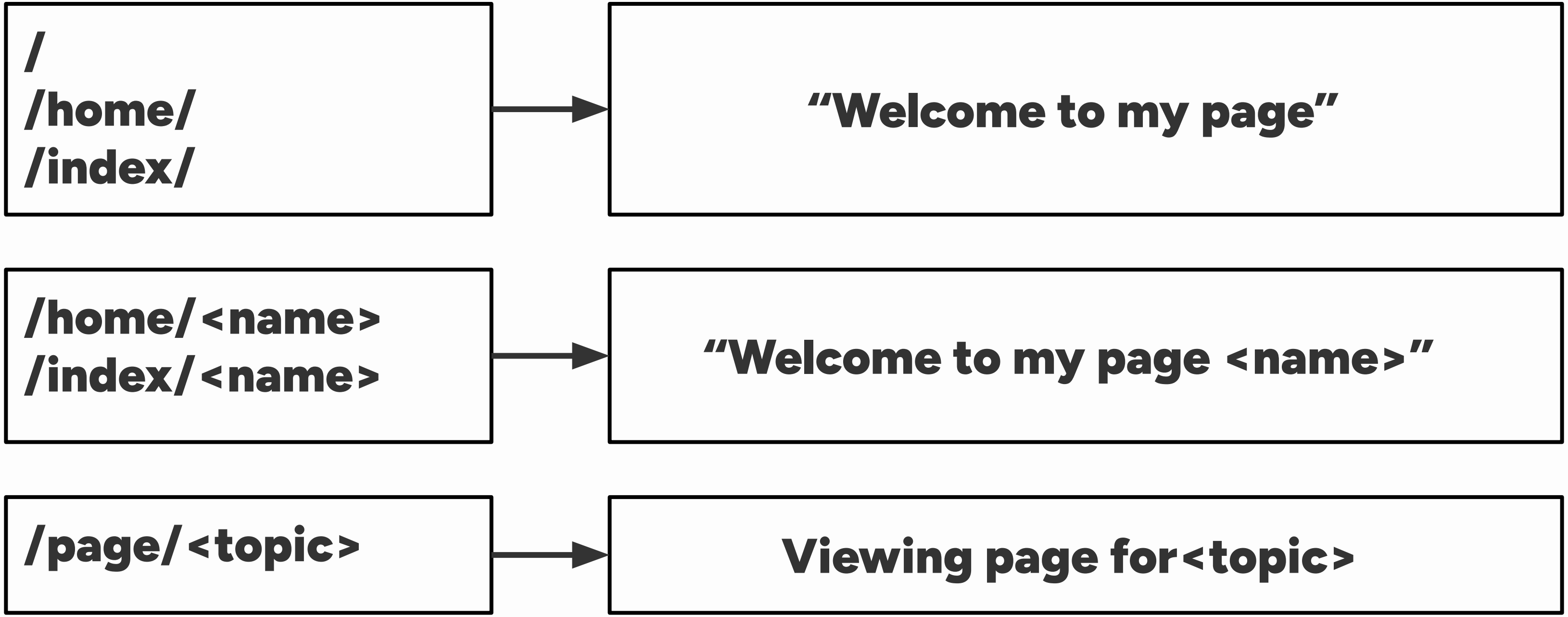
# Dynamic Route

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
     return "Index Page"

@app.route("/profiles/")
def profile():
     return "Profile Page"

@app.route("/profile/<username>")
def dynamic_profile(username):
     return f"Profile {username} Page"

app.run()

```

# Full Dynamic Route

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
     return "Index Page"

@app.route("/profile/")
@app.route("/profiles/")
@app.route("/profile/<username>")
def profile(username=None):
     if username:
          return f"Profile {username} Page"
     else:
          return "Profile Page"

app.run()
```

# Quick Exercise: Provide these routes



/
/home/
/index/

“Welcome to my page”

/home/<name>
/index/<name>

“Welcome to my page <name>”

/page/<topic>

Viewing page for<topic>

# HTML

A crash course on styling text in web pages

# HTML: Hypertext Markup Language

HTML is used to structure and organize content on web pages. It relies on tags, which define elements like headings, paragraphs, and links, to create a webpage’s layout and content.

**<tag>Text</tag>**

**<tag>**

# Headers

Heading tags (**<h1>** to **<h6>**) define the importance and hierarchy of text, with **<h1>** being the highest and **<h6>** the lowest.

```
<h1>Header </h1>
<h2>Header </h2>
<h3>Header </h3>
<h4>Header </h4>
<h5>Header </h5>
<h6>Header </h6>
```

# Headers

Heading tags (**<h1>** to **<h6>**) define the importance and hierarchy of text, with **<h1>** being the highest and **<h6>** the lowest.

<h1> **Header** </h1>

<h2> **Header** </h2>

<h3> **Header** </h3>

<h4> **Header** </h4>

<h5> **Header** </h5>

<h6> **Header** </h6>

# Paragraphs

The <p> tag is used to define paragraphs, separating blocks of text for better readability.

```
<h1>Header </h1>
<p>The p tag is used to define paragraphs   </p>
```

# Paragraphs

The <p> tag is used to define paragraphs, separating blocks of text for better readability.

<h1> Header </h1>

<p> The p tag is used to define paragraphs </p>

# Anchor

The <a> tag is used to create hyperlinks that redirect the user to a different URL.

```
<a href="https://www.example.com   ">Example </a>
```

# Anchor

The **<a>** tag is used to create hyperlinks that redirect the user to a different URL.

<a href="https://www.example.com"> Example </a>

***https://www.example.com***

# Unordered List

The **`<ul>`** tag with **`<li>`** tags enumerate items in bullet point style

```html
<ul>
    <li>First Item</li>
    <li>Second Item</li>
    <li>Third Item</li>
</ul>
```

```
●   First Item
●   Second Item
●   Third Item
```

# Ordered List

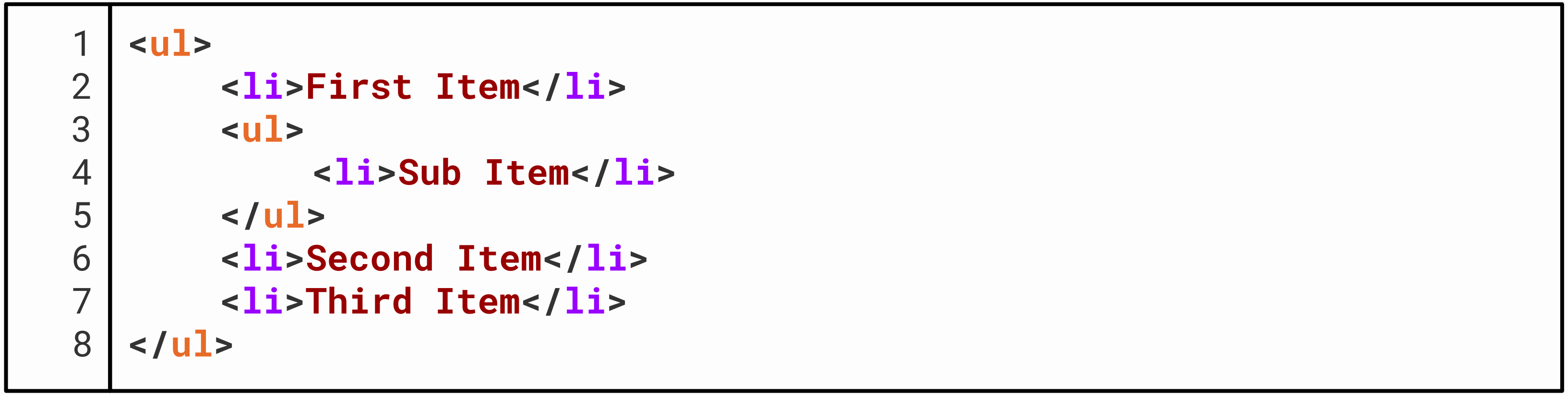
The **<ol>** tag with **<li>** tags enumerate items by number

```
<ol>
    <li>First Item</li>
    <li>Second Item</li>
    <li>Third Item</li>
</ol>
```

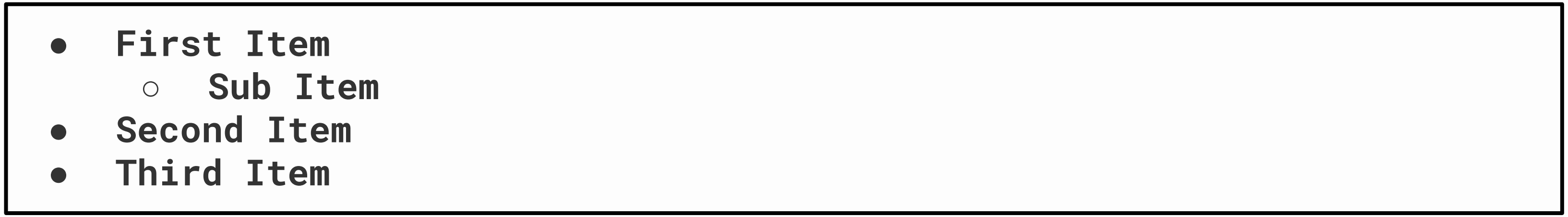
```
1.  First Item
2.  Second Item
3.  Third Item
```

# Nested List

Subitems require an additional tag

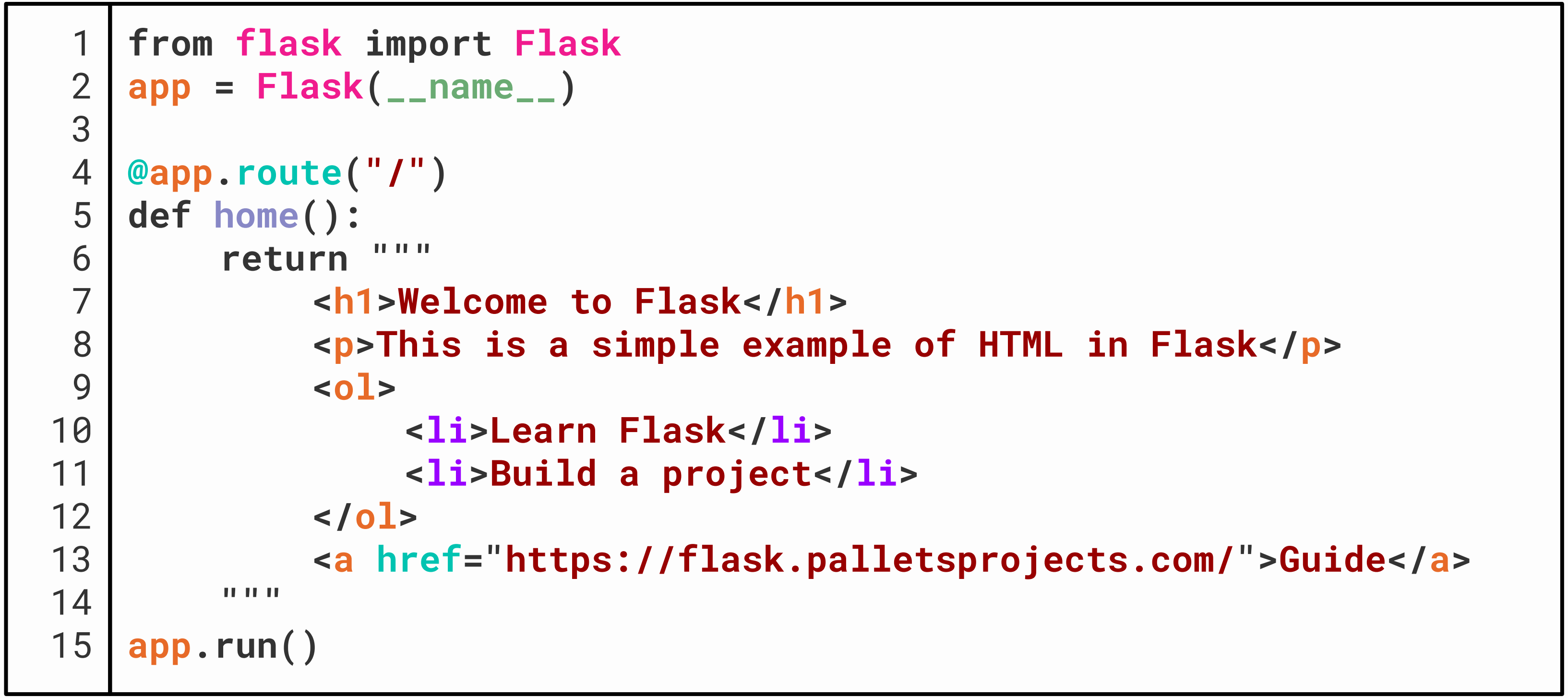


```html
<ul>
    <li>First Item</li>
    <ul>
        <li>Sub Item</li>
    </ul>
    <li>Second Item</li>
    <li>Third Item</li>
</ul>
```



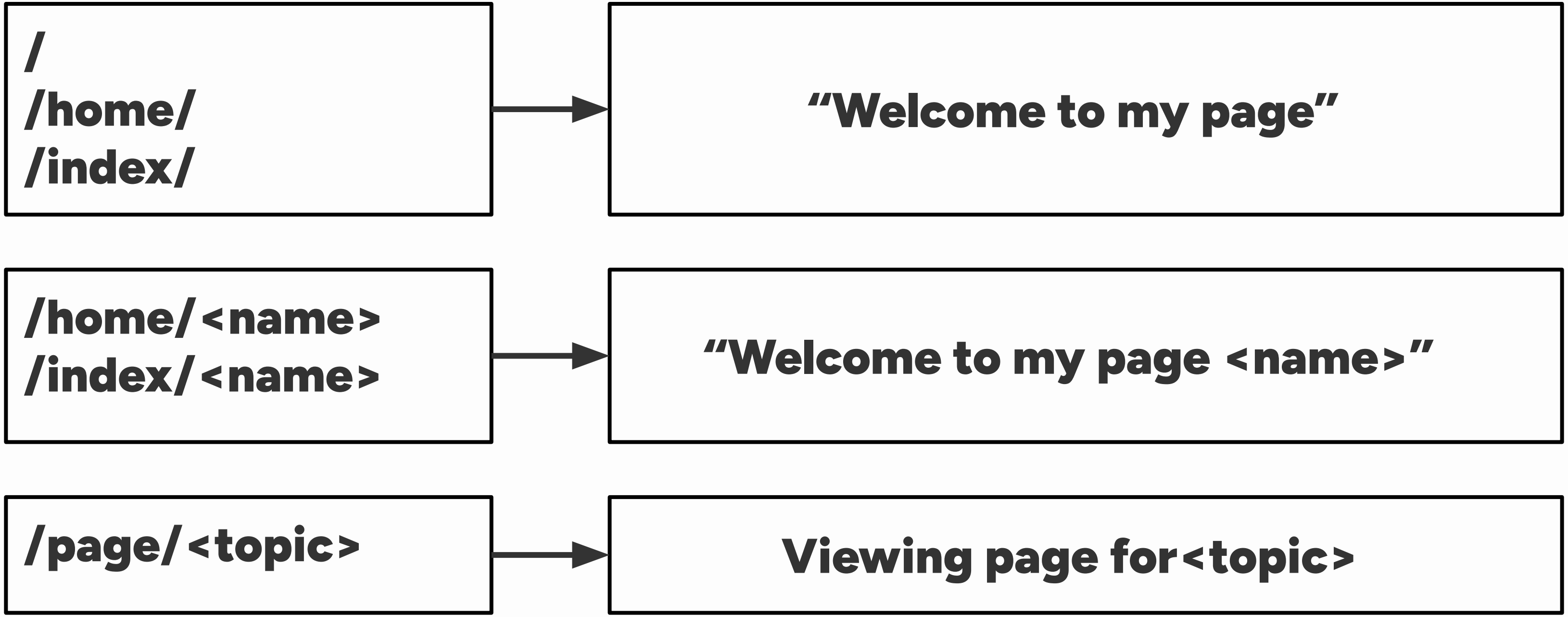
```
●   First Item
    ○   Sub Item
●   Second Item
●   Third Item
```

# HTML Example



```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return """
        <h1>Welcome to Flask</h1>
        <p>This is a simple example of HTML in Flask</p>
        <ol>
            <li>Learn Flask</li>
            <li>Build a project</li>
        </ol>
        <a href="https://flask.palletsprojects.com/">Guide</a>
    """
app.run()
```

# Refactor: Add styling and content



/
/home/
/index/

“Welcome to my page”

/home/<name>
/index/<name>

“Welcome to my page <name>”

/page/<topic>

Viewing page for<topic>

# URL Handling

Special cases for handling subpages

# Dynamic URL

```python
from flask import Flask, url_for
app = Flask(__name__)

@app.route("/")
def index():
    return f'''
    <a href="{url_for('login')}">Login Page</a>
    <a href="{url_for('profile', username=’Ace’)}">Ace</a>
    '''

```

# Dynamic URL

```
@app.route("/login/")
def login():
    return “Login Page”

@app.route("/user/<username>")
def profile(username):
    return f”{username}'s Profile Page”

app.run()

```

# Redirect URL

```python
from flask import Flask, url_for, redirect
app = Flask(__name__)

@app.route("/user/<username>")
def profile(username):
    if username != "admin":
        return redirect(url_for('login'))
    else:
        return "Welcome Admin"

@app.route('/login')
def login():
    return "Please login"

app.run()
```

# Abort Error

```
from flask import Flask, abort

app = Flask(__name__)

@app.route('/')
def index():
    return "Index Page"

@app.route('/login')
def login():
    abort(501)

app.run()
```

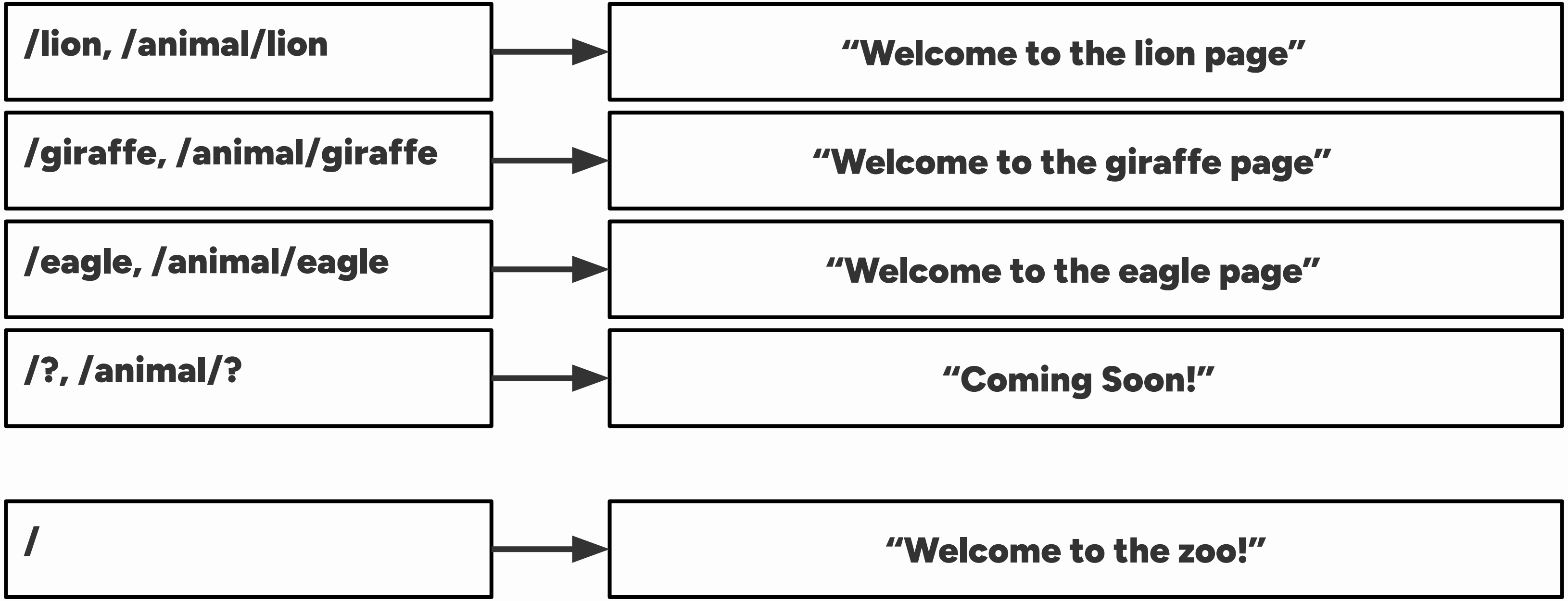
# Error Handler

```python
from flask import Flask, url_for, abort, redirect

app = Flask(__name__)

@app.route("/user/<username>")
def profile(username):
    if username in ['Alex', 'Steve']:
        return f"{username}'s Profile Page"
    elif username == 'Guest':
        return "Guest Profile"
    else:
        abort(401)
```

# Error Handler

```
@app.errorhandler(401)
def handle_401_error(error):
    print(“Undetected visitor”)
    return redirect(url_for('profile', username='Guest'))

app.run()
```

# Quick Exercise: Provide these routes



/lion, /animal/lion → "Welcome to the lion page"

/giraffe, /animal/giraffe → "Welcome to the giraffe page"

/eagle, /animal/eagle → "Welcome to the eagle page"

/?, /animal/? → "Coming Soon!"

/ → "Welcome to the zoo!"

# Requests

Asking users for information

# Login Get

```
from flask import Flask, request

app = Flask(__name__)

@app.get('/login')
def login_get():
    return """
    <form method="post">
        <label for="username">Username:</label>
        <input type="text" name="username">

        <input type="submit">
    </form>
    """
```

# Login Post

```python
@app.post('/login')
def login_post():
    username = request.form['username']
    return f”Form Submitted by {username}”

app.run()
```

# Login Form Get

```python
from flask import Flask, request
app = Flask(__name__)

@app.get('/login')
def login_get():
    return """
    <form method="post">
        <label for="username">Username:</label>
        <input type="text" name="username"><br>
        <label for="password">Password:</label>
        <input type="password" name="password”><br>
        <label for="email">Email:</label>
        <input type="email" name="email"><br>
        <input type="submit" value="Login">
    </form>
    """
```

# Login Form Post

```
def valid(username, email , password ):
    return not (
        username == "admin"
        and password == "pass"
        and email == "admin@gmail.com"
    )
@app.post('/login')
def login_post():
    username = request.form['username']
    password = request.form['password']
    email = request.form['email']
    If not valid(username, email , password ):
        return 'Invalid credentials!'
    else:
        return 'Login successful!'
```

# Sessions

Server-side data storage

# Session Setup

```python
from flask import Flask, request, redirect, url_for, session

app = Flask(__name__)
app.secret_key = 'your_secret_key'

users = {
    "admin": "password123",
    "user": "pass456"
}

```

# Session Home

```
@app.route('/')
def home():
    if 'username' in session:
        return f”””
            Welcome, {session['username']}!
            <a href='/logout'>Logout</a>
        ”””
    else:
        return f”””
            Welcome!
            <a href='/login'>Login</a>
        ”””
```

# Session Login Get

```
@app.get('/login')
def login_get():
    return f"""
    <form method="post">
        <label for="username">Username:</label>
        <input type="text" name="username"><br>
        <label for="password">Password:</label>
        <input type="password" name="password"><br>
        <input type="submit" value="Login">
    </form>
    """

```

# Session Validation

```
@app.post('/login')
def login_post():
    username = request.form['username']
    password = request.form['password']
    if username in users and users[username] == password :
        session['username'] = username
        return redirect(url_for('home'))
    else:
        return redirect(url_for('login_get'))

@app.route('/logout')
def logout():
    session.pop('username', None)
    return redirect(url_for('home'))

app.run()
```

# Templates

Adding placeholders and logic to HTML

# Render Template

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

app.run()
```

# Render Template - HTML

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Demo App</title>
    </head>

    <body>
        <h1>Demo Page</h1>
        <p>Simple demo application</p>
    </body>
</html>

```

# Render Template - Parameter

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template(
        ”index_variable.html”,
        title="Template App",
        message="Template Demo Page",
        additional_message="Template used",
    )

app.run()
```

# Render Template - HTML Parameter

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>{{ title }}</title>
    </head>

    <body>
        <h1>{{ message }}</h1>
        <p>This is a simple Flask demo application</p>
        {{ additional_message }}
    </body>
</html>

```

# Render Template - Conditional

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template(conditional.html', logged_in=True)

app.run()
```

# Render Template - HTML Conditional

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Login</title>
    </head>
    <body>
        {% if logged_in %}
            <p>Welcome back, user!</p>
        {% else %}
            <p>Please log in to continue.</p>
        {% endif %}
    </body>
</html>
```

# Render Template - Items

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    items = ['Apple', 'Banana', 'Cherry']
    return render_template(‘items.html', items=items)

app.run()
```

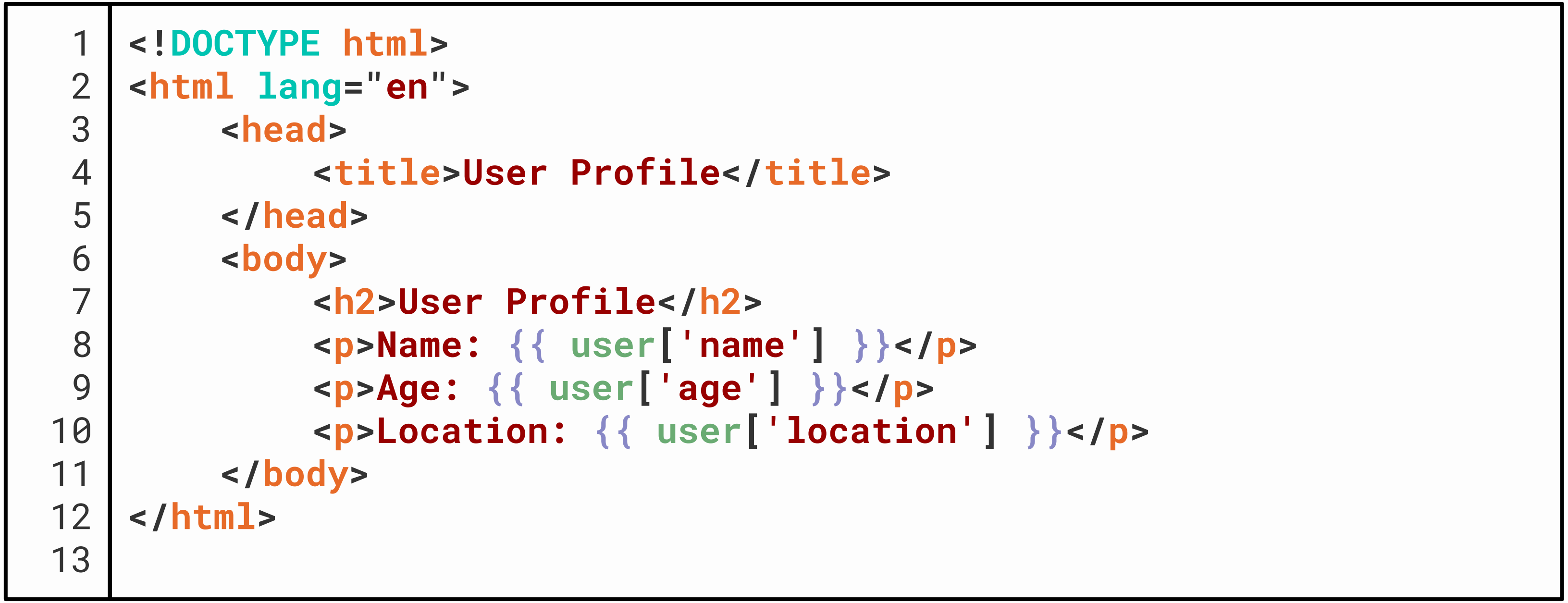
# Render Template - HTML Loop

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Items</title>
    </head>
    <body>
        <h2>Available Items:</h2>
        <ul>
        {% for item in items %}
            <li>{{ item }}</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

# Render Template - Dictionary

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    user_info = {
        'name': 'Eren',
        'location': 'Manila'
    }
    return render_template(‘profiles.html', user=user_info)

app.run()
```

# Render Template - HTML Dictionary

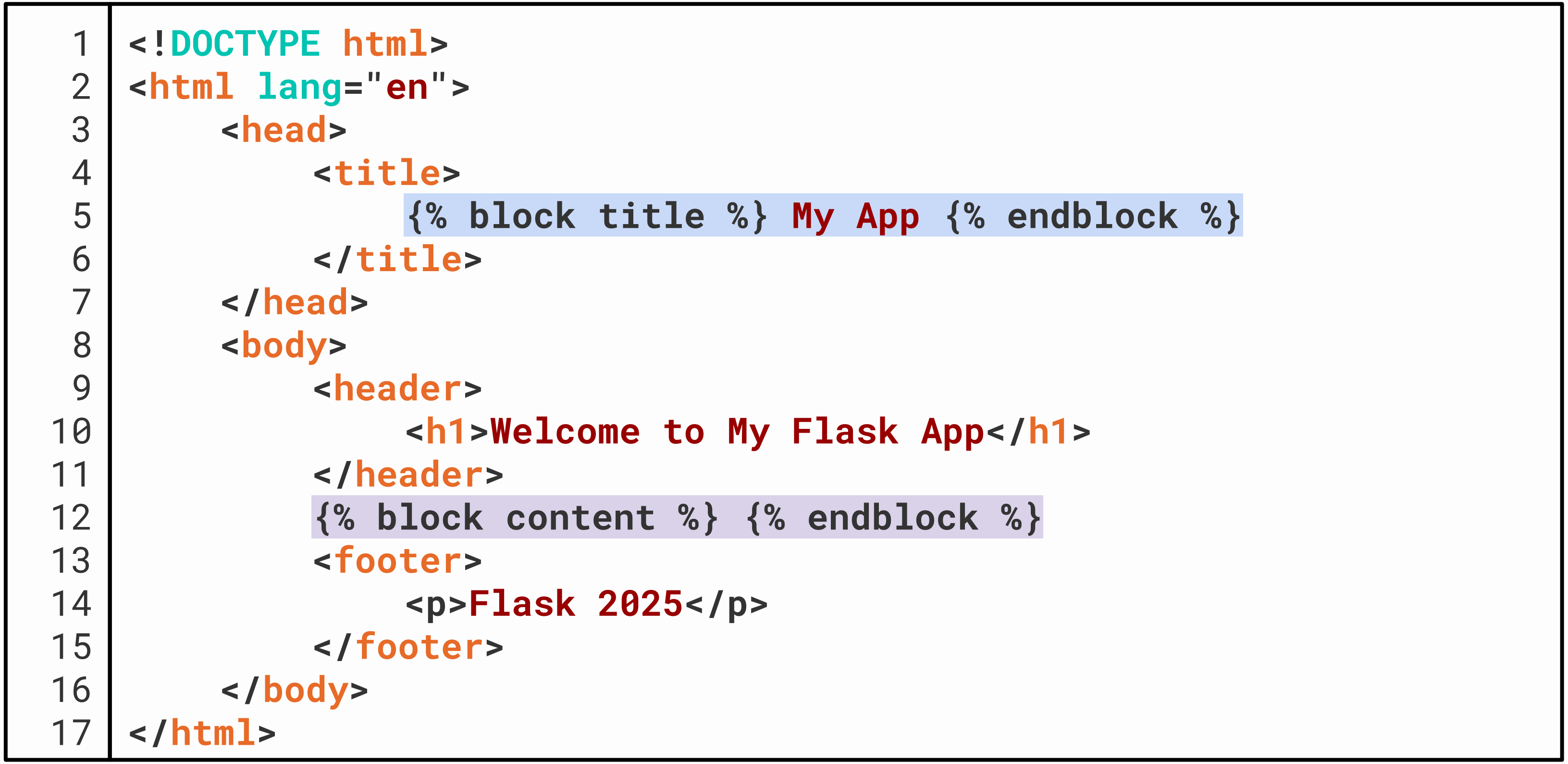


```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>User Profile</title>
    </head>
    <body>
        <h2>User Profile</h2>
        <p>Name: {{ user['name'] }}</p>
        <p>Age: {{ user['age'] }}</p>
        <p>Location: {{ user['location'] }}</p>
    </body>
</html>

```

# Components

Templating the HTML files themselves

# Parent HTML



```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>
            {% block title %} My App {% endblock %}
        </title>
    </head>
    <body>
        <header>
            <h1>Welcome to My Flask App</h1>
        </header>
        {% block content %} {% endblock %}
        <footer>
            <p>Flask 2025</p>
        </footer>
    </body>
</html>
```

# Child HTML

```
{% extends 'parent.html' %}

{% block title %}
    Home
{% endblock %}

{% block content %}
    <h1>Subclass Page</h1>
    <p>Welcome to the subclass page!</p>
{% endblock %}
```

# OpenPyXL

Lightweight library for reading xlsx and xlsm files

# Excel Basics

Common Read-Write Operations for Excel Files

# Creating a Workbook

In OpenPyXL, an entire Excel file is represented using the **Workbook** class. All of the data processes (loading, saving, editing), sheet handling, and cell management is done here.

```
from openpyxl import Workbook

workbook = Workbook()



workbook.save("sample.xlsx")
```

# Default Worksheet

Accessing a worksheet is done using indexing. By default, a new workbook has a starting sheet with the title "Sheet"

```python
from openpyxl import Workbook

workbook = Workbook()
sheet = workbook["Sheet"]


workbook.save("sample.xlsx")
```

# Creating a Worksheet

A Workbook object can use the create_sheet(str) method to create a new sheet. It gets added at the end by default. If you want to set the index, use create_sheet(str, int).

```python
from openpyxl import Workbook

workbook = Workbook()
sheet = workbook["Sheet"]
workbook.create_sheet("Additional")

workbook.save("sample.xlsx")
```

# Editing a Cell

Accessing a worksheet is done using indexing. The key depends on the coordinate used in Excel workbooks

```python
from openpyxl import Workbook

workbook = Workbook()
sheet = workbook["Sheet"]
workbook.create_sheet("Additional")
sheet["A1"] = "Hello"
workbook.save("sample.xlsx")
```

# Loading a Workbook

You can also load existing Excel files using the `load_workbook` helper function.

```python
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
```

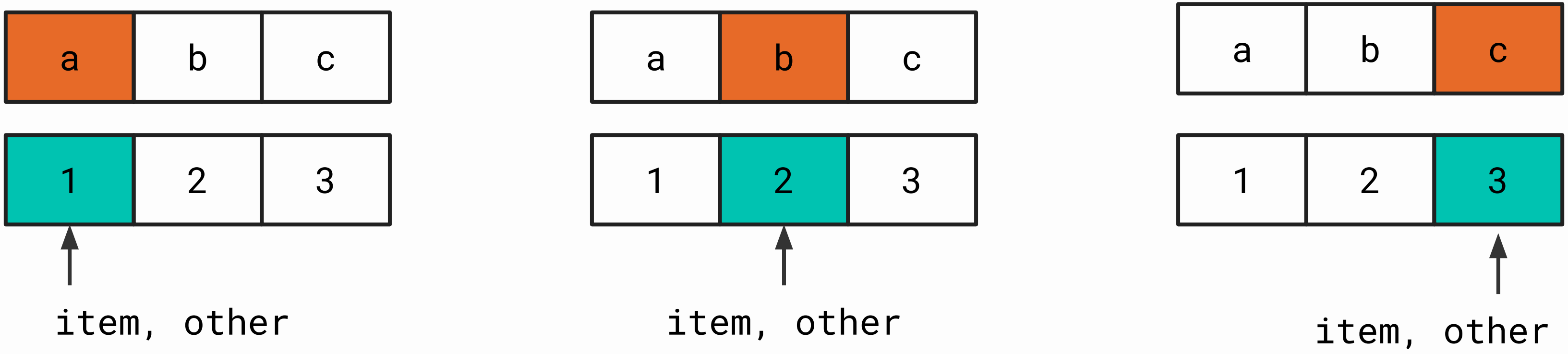
# Recap: Multi-Loop

Recall the mechanics of zip, enumerate, and tuple

# Multiple Looping

You can access two items at once from two different sequences using the zip function

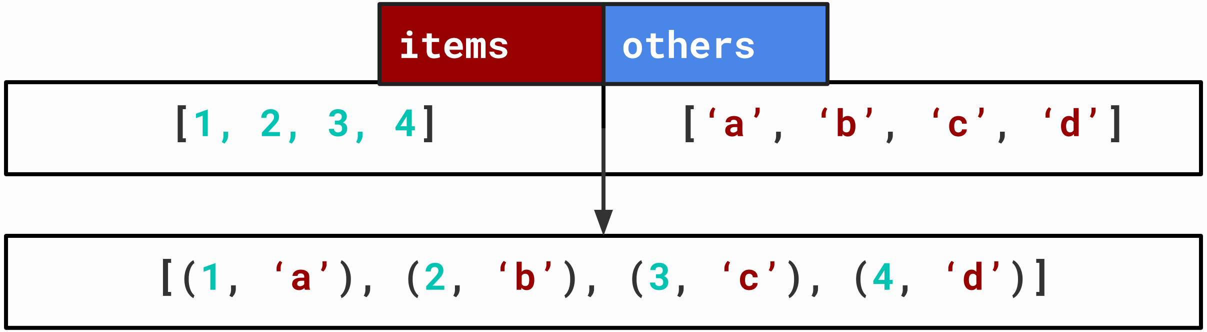
```
items = (‘a’, ‘b’, ‘c’)
others = (1, 2, 3)
for item, other in zip(items, others):
     print(item, other)
```



# Zip Function Contents

The **zip** function creates a list of tuples from all of its parameters

```
items = (‘a’, ‘b’, ‘c’)
others = (1, 2, 3)
zipped = zip(items, others)
print(list(zipped))
```



[1, 2, 3, 4]

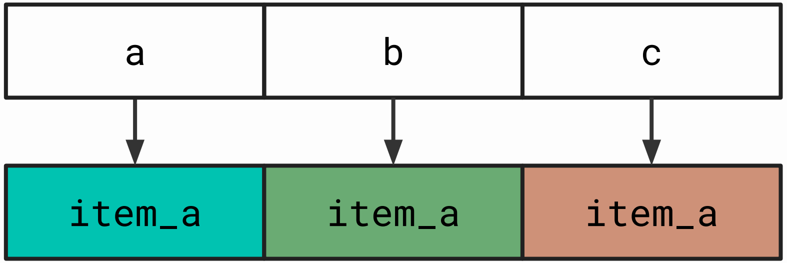
[‘a’, ‘b’, ‘c’, ‘d’]

[(1, ‘a’), (2, ‘b’), (3, ‘c’), (4, ‘d’)]

# Tuple Unpacking

Because tuples have a fixed size, Python added an unpacking feature for convenience

```
items = (‘a’, ‘b’, ‘c’)
item_a, item_b, item_c = items
```



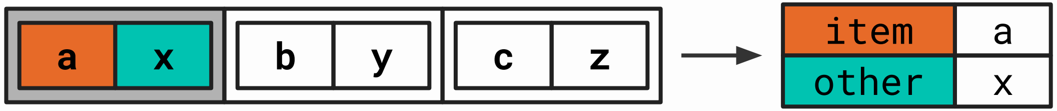
a b c

item_a item_a item_a

# Unpacking in Loops

You can access two items at once from two different sequences using the zip function

```
items = (‘a’, ‘b’, ‘c’)
others = (‘x’, ‘y’, ‘z’)
for item, other in zip(items, others):
    print(item, other)
```

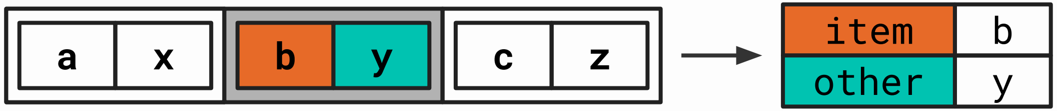


a x | b y | c z → item a, other x

# Unpacking in Loops

You can access two items at once from two different sequences using the zip function

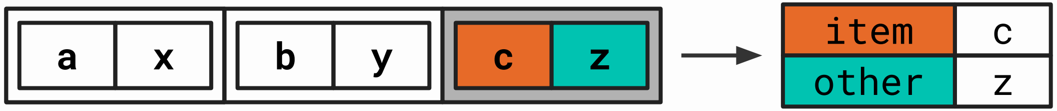
```
items = (‘a’, ‘b’, ‘c’)
others = (‘x’, ‘y’, ‘z’)
for item, other in zip(items, others):
    print(item, other)
```



# Unpacking in Loops

You can access two items at once from two different sequences using the zip function

```
items = (‘a’, ‘b’, ‘c’)
others = (‘x’, ‘y’, ‘z’)
for item, other in zip(items, others):
    print(item, other)
```



a x | b y | c z

| | |
|---|---|
| item | c |
| other | z |

# Enumerate Looping

You can loop through a sequence of items and get their position using the enumerate function.

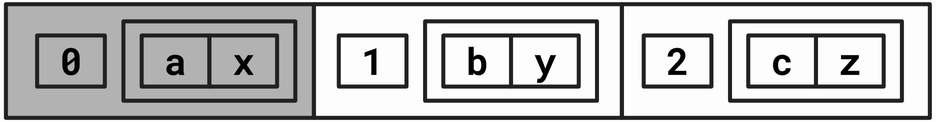
```
items = (‘a’, ‘b’, ‘c’)
for index, item in enumerate(items):
    print(index, item)
```

```
0 a
1 b
2 c
```

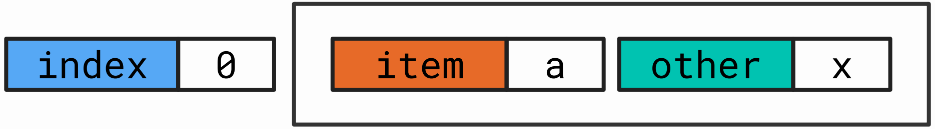
# Nested Unpacking

For inner tuples inside another tuple, denote using parentheses

```
items = (‘a’, ‘b’, ‘c’)
others = (‘x’, ‘y’, ‘z’)
for index, (items, other) in enumerate(zip(items, others)):
    print(item, other)
```



0 a x 1 b y 2 c z

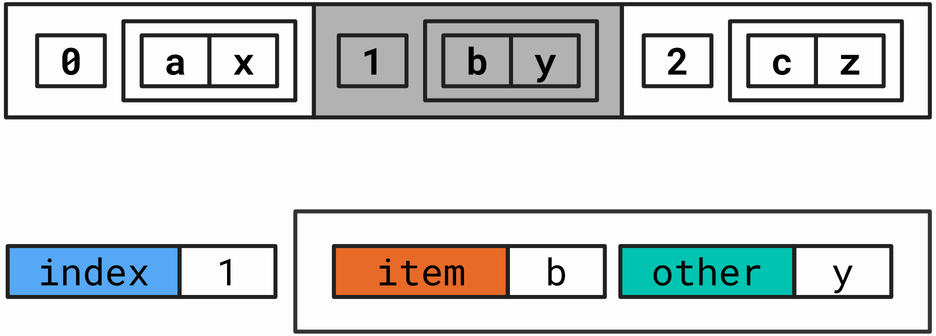


index 0 item a other x

# Nested Unpacking

For inner tuples inside another tuple, denote using parentheses

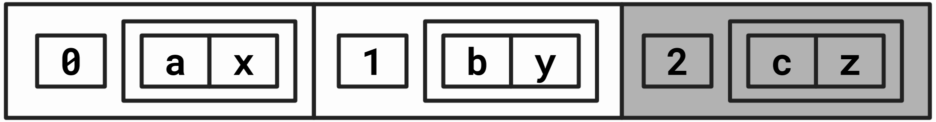
```
items = (‘a’, ‘b’, ‘c’)
others = (‘x’, ‘y’, ‘z’)
for index, (items, other) in enumerate(zip(items, others)):
    print(item, other)
```



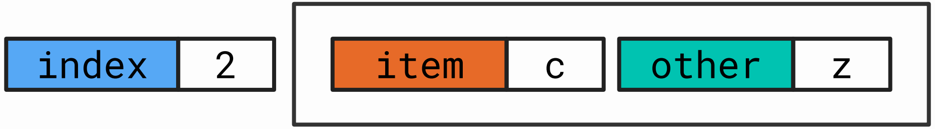
# Nested Unpacking

For inner tuples inside another tuple, denote using parentheses

```
items = (‘a’, ‘b’, ‘c’)
others = (‘x’, ‘y’, ‘z’)
for index, (items, other) in enumerate(zip(items, others)):
    print(item, other)
```



0 a x 1 b y 2 c z



index 2 item c other z

# Pair Unpacking

For inner tuples inside another tuple, denote using parentheses

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'a': 10, 'b': 20}

for (k1, v1), (k2, v2) in zip(dict1.items(), dict2.items()):
    print(k1, v1, k2, v2)
```

# Cell Management

Example operations and methods for cell read and writes

# Read-Write Cells

Cells inside worksheets can either be accessed using indexing or the **Cell** interface.

```python
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

sheet["A1"] = "Tickets"
print(sheet["A1"].value)

cell = sheet.cell(row=1, column=2)
cell.value = 100
print(cell.value)

workbook.save("sample.xlsx")
```

# Multiple Cell Write

There is no dedicated method for writing in multiple cells at once. Instead, the expected approach is to use a standard loop

```python
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

tickets = {"HR": 30, "Legal": 23, "Sales": 34, "Admin": 13}

for i, (group, count) in enumerate(tickets.items(), start=3):
    sheet.cell(row=i, column=1).value = group
    sheet.cell(row=i, column=2).value = count

workbook.save("sample.xlsx")
```

# Multiple Cell Write (Ranges)

Worksheets support Excel-based formulas for getting items. This allows cell-based coding.

```
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

tickets = {"HR": 30, "Legal": 23, "Sales": 34, "Admin": 13}

ticket_and_cells = zip(tickets.items(), sheet["A3:B6"])

for (group, count), (group_cell, count_cell) in ticket_and_cells:
    group_cell.value = group
    count_cell.value = count

workbook.save("sample.xlsx")
```

# Multiple Cell Append

While OpenPyXL doesn’t support writing on ranges directly, it allows appends.

```python
from openpyxl import load_workbook
workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

new_data = ["Tech", 300]
sheet.append(new_data)

workbook.save("sample.xlsx")
```

# Multiple Cell Read

Each **Worksheet** object has an `iter_rows` method to loop or iterate through all of the cells. Each row is a tuple of **Cell** objects.

```python
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

for row in sheet.iter_rows():
    print(row)
```

# Multiple Cell Read (Unpacked)

If there are only a few number of columns, you can directly assign the values to variables similar to how **enumerate** and **zip** operates.

```python
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

for header, item in sheet.iter_rows():
    print(header.value, item.value)
```

# Multiple Cell Read (Bounded)

The **`iter_rows`** method can change where it starts and ends using the min_row, and max_col optional parameters. The default is the first row and the last row with a value.

```
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

for header, item in sheet.iter_rows(min_row=3, max_row=6):
    print(header.value, item.value)
```

```
tip: you can use sheet.max_row and max.column
```

# Quick Exercise: Product Orders

Create a new sheet called **`Order`** in **`samples.xlsx`** and generate the following data

| **Category** | **Brand** | **Unit** |
|---|---|---|
| ***Laptop*** | HP | 1 |
| ***Laptop*** | HP | 2 |
| ***Laptop*** | Acer | 3 |
| ***Laptop*** | Acer | 4 |
| ***Monitor*** | HP | 1 |
| ***Monitor*** | HP | 2 |
| ***Monitor*** | Acer | 3 |
| ***Monitor*** | Acer | 4 |

# Cell+

Adding styling and rules for the cell layouts

# Cell Font

`Cell` objects have the `font` property that can be changed to add font-specific styling

```python
from openpyxl import load_workbook
from openpyxl.styles import Font

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

sheet["A1"].font = Font(name="Arial", size=20)
workbook.save("sample.xlsx")
```

# Cell Font (Options)

`Cell` objects have the `font` property that can be changed to add styling

| Property | Description |
|---|---|
| name | `‘Calibri’, ‘Arial’, ‘Times New Roman’, etc. (system-based)` |
| size | `float/int` |
| bold | `bool` |
| italic | `bool` |
| underline | `‘single’, ‘double’, ‘singleAccounting’, ‘doubleAccounting’, None/False` |
| strike | `bool` |
| color | **`Hex Codes:`** `‘FF0000’ (Red), ‘00FF00’ (Green), ‘000000’ (Black), etc.` |

# Cell Pattern Fill

`Cell` objects have the `fill` property that can be changed to add background styling

```python
from openpyxl import load_workbook
from openpyxl.styles import PatternFill

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

for (cell,) in sheet["A3:A7"]:
    cell.fill = PatternFill(fill_type='solid', fgColor='4F81BD')

workbook.save("sample.xlsx")
```

# Cell Pattern Border and Side

`Cell` objects have the `border` property that can be changed to add border styling

```python
from openpyxl import load_workbook
from openpyxl.styles import Side, Border

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

ss = Side(style="thin", color='000000')

for (cell,) in sheet["A3:A7"]:
    cell.border = Border(left=ss, right=ss, top=ss, bottom=ss)

workbook.save("sample.xlsx")
```

# Cell Side (Options)

Side objects have the following styles to choose from

| Property | Description |
|---|---|
| style | 'thin', 'medium', 'thick', 'dashed', 'dotted', 'double', 'hair', 'mediumDashed', 'slantDashDot' |
| color | **Hex Codes:** ‘FF0000’ (Red), ‘00FF00’ (Green), ‘000000’ (Black), etc. |

# Cell Alignment

`Cell` objects have the `alignment` property that can be changed for text formatting

```python
from openpyxl import load_workbook
from openpyxl.styles import Alignment

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

for (cell,) in sheet["A3:A7"]:
    cell.alignment = Alignment(
        horizontal='center', vertical='center',
        wrap_text=True, shrink_to_fit=True,
        indent=1
    )

workbook.save("sample.xlsx")
```

# Cell Alignment (Options)

The properties in the `Alignment` class have the following options

| Property | Description |
|---|---|
| horizontal | `'left', 'right', 'center', 'justify'` |
| vertical | `'top', 'center', 'bottom'` |

# Cell Number Format

`Cell` objects have the `alignment` property that can be changed for text formatting

```python
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]

sheet["B1"].number_format = '#,##0'
workbook.save("sample.xlsx")
```

| | |
|---|---|
| Date Format | `'mm/dd/yyyy'` |
| Time | `'hh:mm:ss'` |
| Percentage | `'0%'` |
| Decimal | `'0.00'` |

# Quick Exercise: Product Orders (Styled)

Follow the styling below for the `Order` sheet in `samples.xlsx`

| **Category** | **Brand** | **Unit** |
|---:|---:|---:|
| ***Laptop*** | HP | 1 |
| ***Laptop*** | HP | 2 |
| ***Laptop*** | Acer | 3 |
| ***Laptop*** | Acer | 4 |
| ***Monitor*** | HP | 1 |
| ***Monitor*** | HP | 2 |
| ***Monitor*** | Acer | 3 |
| ***Monitor*** | Acer | 4 |

# Protection

Adding write safety to the worksheet

# Sheet Protection (Specific)

```python
from openpyxl import load_workbook


workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]
sheet.protection.sheet = True




workbook.save("secured.xlsx")
```

# Sheet Protection (Specific)

```python
from openpyxl import load_workbook
from openpyxl.styles import Protection

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]
sheet.protection.sheet = True

for (cell,) in sheet["B2:B7"]:
    cell.protection = Protection(locked=False)

workbook.save("secured.xlsx")
```

# Data Validation (Contains)

Category-based (finite type of strings) can be limited using the `DataValidation` class

```python
from openpyxl import load_workbook
from openpyxl.worksheet.datavalidation import DataValidation

workbook = load_workbook("sample.xlsx")
sheet = workbook["Order"]

options_str = '"Laptop,Monitor,Peripheral"'
dv = DataValidation(type="list", formula1=options_str)

sheet.add_data_validation(dv)
dv.add("A2:A100")
workbook.save("sample.xlsx")
```

# Deletion

How to remove or clear out values

# Sheet Deletion

Remove a sheet can be done directly using the **`del`** operator

```
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
del workbook["Sheet"]

workbook.save("sample.xlsx")
```

# Cell Deletion

There is no direct way to delete cells since it works on a reference basis but you can clear it

```python
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]
sheet["A1"] = None
sheet["B1"] = None

workbook.save("sample.xlsx")
```

# Row Deletion

There is no direct way to delete cells since it works on a reference basis but you can clear it

```python
from openpyxl import load_workbook

workbook = load_workbook("sample.xlsx")
sheet = workbook["Additional"]
sheet.delete_rows(1)
sheet.delete_rows(1)

workbook.save("sample.xlsx")
```

# Quick Exercise: Dummy Logs

Create a new workbook **`tickets.xlsx`**. In sheet **`Tickets`**, create `10_000` random entries

```python
from random import randint, choice, seed
from datetime import datetime, timedelta

seed(123)

# Example of how to generate random values for a row
status = choice(["New", "Ongoing", "Done", "Close”, None])
priority = choice(["Low", "Medium", "High", None])
department = choice(["HR", "Legal", "sales ", "Adm", "Tech"])
points = randint(1, 100)
votes = randint(1, 10)
start = datetime(2023, 5, 1) + timedelta(hours=randint(0, 2000))
end = start + timedelta(hours=randint(0, 2000))
```

# Quick Exercise: Dummy Accounts

Create a new workbook **accounts.xlsx**. In sheet **Logs** create 10_000 random entries

```
from random import randint, choice, seed
from datetime import datetime, timedelta

seed(123)

# Example of how to generate random values for a row
accounts = choice([...])
sector = choice([...])
year_established = randint(1900, 2025)
revenue = randint(10_000, 100_000_000_000)
employees = randint(1, 1_000_000)
office_location = choice([...])
subsidiary_of = choice([...])
```

# Pandas

The most common technique for tabular data manipulation

# Reading Data

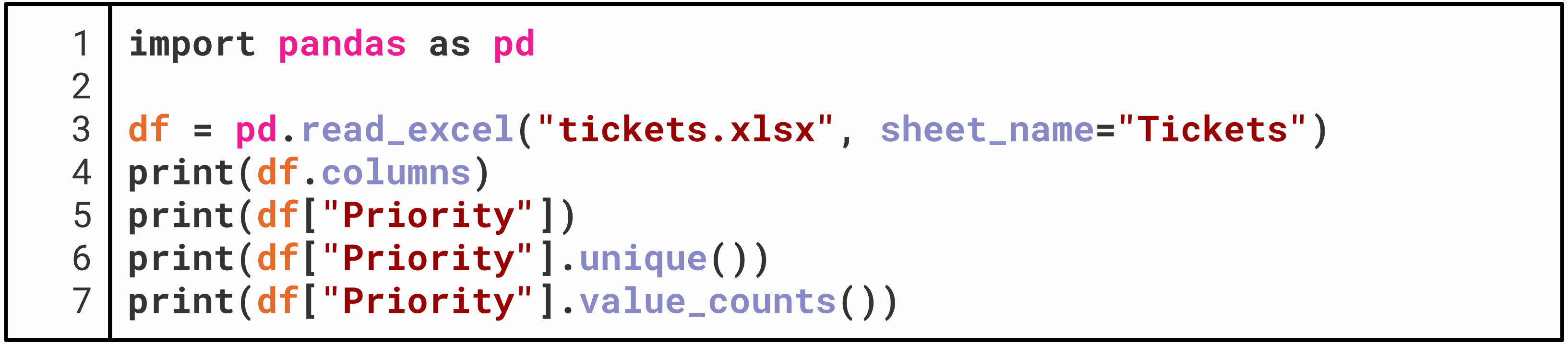
Pandas converts tabular data to data frames that are convenient to read and access

```python
import pandas as pd

df = pd.read_csv("tickets.csv")
print(df)
print(df.info())
print(df.describe())
```

```python
import pandas as pd

df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
print(df)
print(df.info())
print(df.describe())
```

# Dataframe Columns

Pandas makes column access very convenient using the indexing operation



```python
import pandas as pd

df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
print(df.columns)
print(df["Priority"])
print(df["Priority"].unique())
print(df["Priority"].value_counts())
```

# Dataframe New Columns

Pandas specializes in creating new columns using data from other columns

```python
import pandas as pd

df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")

df["Duration"] = df["End"] - df["Start"]
df["Duration"] = df["Duration"].dt.total_seconds()
df["Duration"] = df["Duration"] / 3600

print(df)
```

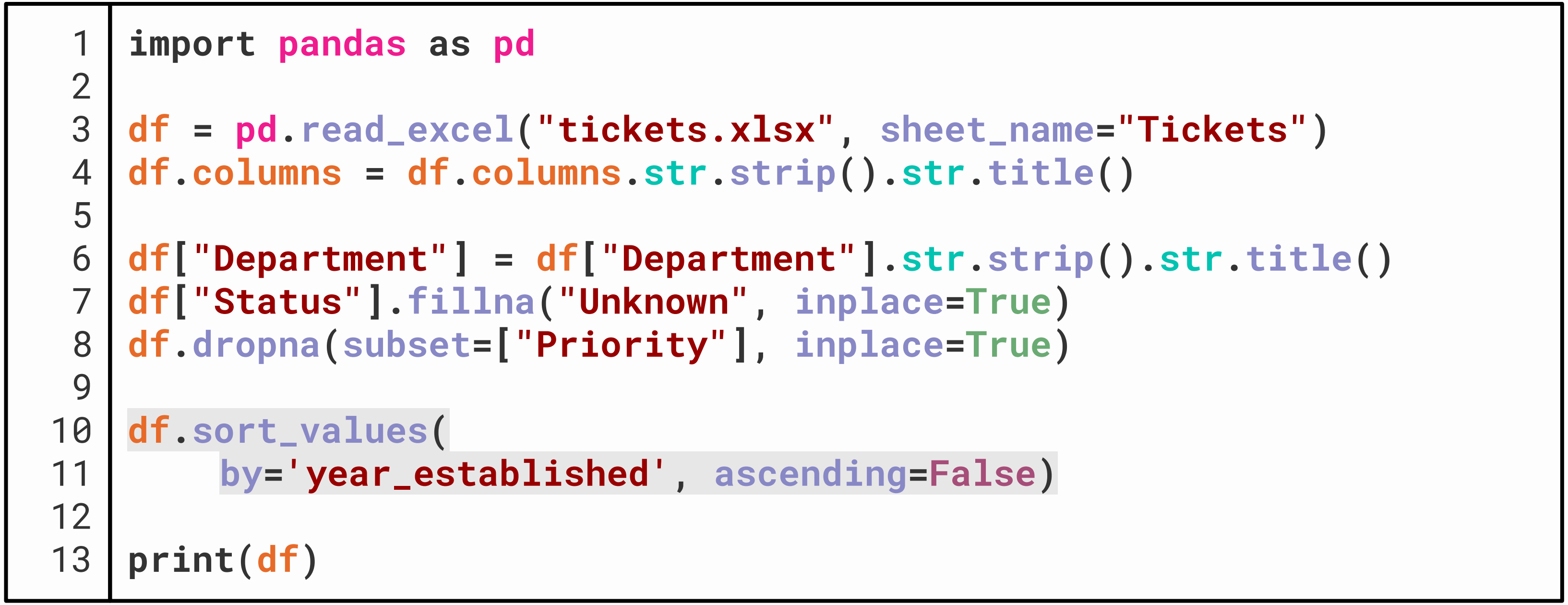
# Data Processes

Common operations and methods for data preparation

# Common Data Cleaning Techniques

```python
import pandas as pd

df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
df.columns = df.columns.str.strip().str.title()

df["Department"] = df["Department"].str.strip().str.title()
df["Status"].fillna("Unknown", inplace=True)
df.dropna(subset=["Priority"], inplace=True)

print(df)
```

# Sorting by Column



```python
import pandas as pd

df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
df.columns = df.columns.str.strip().str.title()

df["Department"] = df["Department"].str.strip().str.title()
df["Status"].fillna("Unknown", inplace=True)
df.dropna(subset=["Priority"], inplace=True)

df.sort_values(
    by='year_established', ascending=False)

print(df)
```

# Saving in a New Excel File

```python
import pandas as pd

df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
df.columns = df.columns.str.strip().str.title()

df["Department"] = df["Department"].str.strip().str.title()
df["Status"].fillna("Unknown", inplace=True)
df.dropna(subset=["Priority"], inplace=True)

df.sort_values(
    by='year_established', ascending=False)

print(df)
df.to_excel("tick_new.xlsx", sheet_name="Tickets", index=False)
```

# Appending to an Existing Excel File

```python
import pandas as pd

df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
df.columns = df.columns.str.strip().str.title()

df["Department"] = df["Department"].str.strip().str.title()
df["Status"].fillna("Unknown", inplace=True)
df.dropna(subset=["Priority"], inplace=True)

df.sort_values(
    by='year_established', ascending=False)

print(df)
with pd.ExcelWriter('tickets.xlsx', mode='a') as writer:
    df.to_excel(writer, sheet_name="Clean Tickets", index=False)
```

# Pandas Filtering

```python
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")

high_revenue = df[df['Revenue'] > 100_000_000]
tech_sector = df[df['Sector'] == "Technology"]

print(df)
with pd.ExcelWriter('accounts.xlsx', mode='a') as writer:
    tech_sector.to_excel(writer, sheet_name="Tech", index=False)
    high_revenue.to_excel(writer, sheet_name="Top", index=False)
```

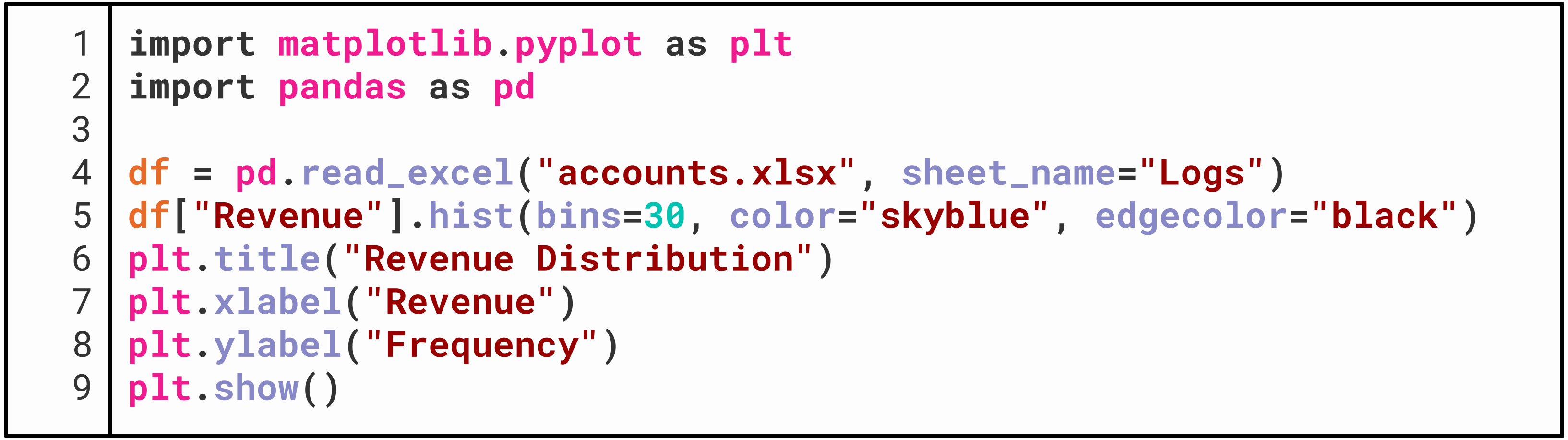
# Grouping and Aggregation

```
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")

avg_revenue = df.groupby('Sector')['Revenue'].mean()
total_employees = df.groupby('Sector')['Employees'].sum()
sector_count = df['Sector'].value_counts()

print('Average Revenue', avg_revenue)
print('Total Employees', total_employees)
print('Sector Count', sector_count)
```

# Data Visualization

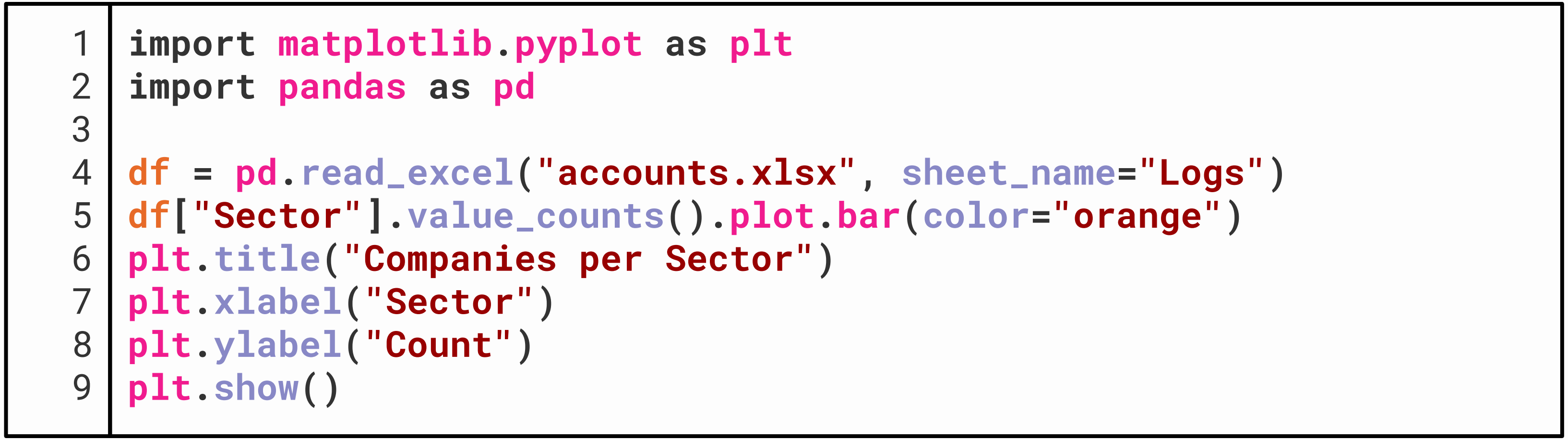
Examples of all visualizations

# Histogram (Number Distribution)



```python
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
df["Revenue"].hist(bins=30, color="skyblue", edgecolor="black")
plt.title("Revenue Distribution")
plt.xlabel("Revenue")
plt.ylabel("Frequency")
plt.show()
```

# Bar Chart (Change Over Unit)



```python
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
df["Sector"].value_counts().plot.bar(color="orange")
plt.title("Companies per Sector")
plt.xlabel("Sector")
plt.ylabel("Count")
plt.show()
```

# Scatter Plot Chart (Spatial Relationship)

```python
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
df["Office Location"].value_counts().head(5).plot.pie()
plt.title("Top 5 Office Locations (Share)")
plt.xlabel("Sector")
plt.ylabel("")
plt.show()
```

# Pie Chart (Percent Composition)

```python
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
df["Office Location"].value_counts().head(5).plot.pie()
plt.title("Top 5 Office Locations (Share)")
plt.xlabel("Sector")
plt.ylabel("")
plt.show()
```

# Box Plot (Statistics Summary)

```python
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
df.boxplot(column="Revenue", by="Sector")
plt.title("Revenue Distribution by Sector")
plt.xlabel("Sector")
plt.ylabel("Revenue")
plt.tight_layout()
plt.show()
```

# Line Plot (Change Over Unit)

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
df.groupby("Year Established")["Revenue"].mean().plot.line()
plt.title("Average Revenue by Year Established")
plt.xlabel("Year")
plt.ylabel("Average Revenue")
plt.show()
```

# Stacked Bar Chart (Composition + Growth)

```python
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
stack_data = df.groupby(["Year Established", "Sector"])
stack_data = stack_data.size().unstack().fillna(0)

stack_data.plot.bar(stacked=True)
plt.title("Companies per Year by Sector")
plt.xlabel("Year Established")
plt.ylabel("Company Count")
plt.tight_layout()
plt.show()
```

# Streamlit

Modern web app framework for simple, data-driven use cases

# Virtual Environments

Prerequisite for using Streamlit if not in PyCharm

# Virtual Environment

A virtual environment (venv) isolates packages for your project from the entire system. This prevents package conflicts, prevents clutter, and makes the project reproducible. The following code creates a folder .venv that will store isolated packages

### Windows

```
$ python -m venv .venv
```

### Linux/MacOS

```
$ python3 -m venv .venv
```

# Virtual Environment - Activation

To actually use the packages of a virtual environment, you need to **`activate`** it first.

### Windows (Command Prompt)

```
$ .venv\Scripts\activate
```

### Windows (Powershell)

```
$ .venv\Scripts\Activate.ps1
```

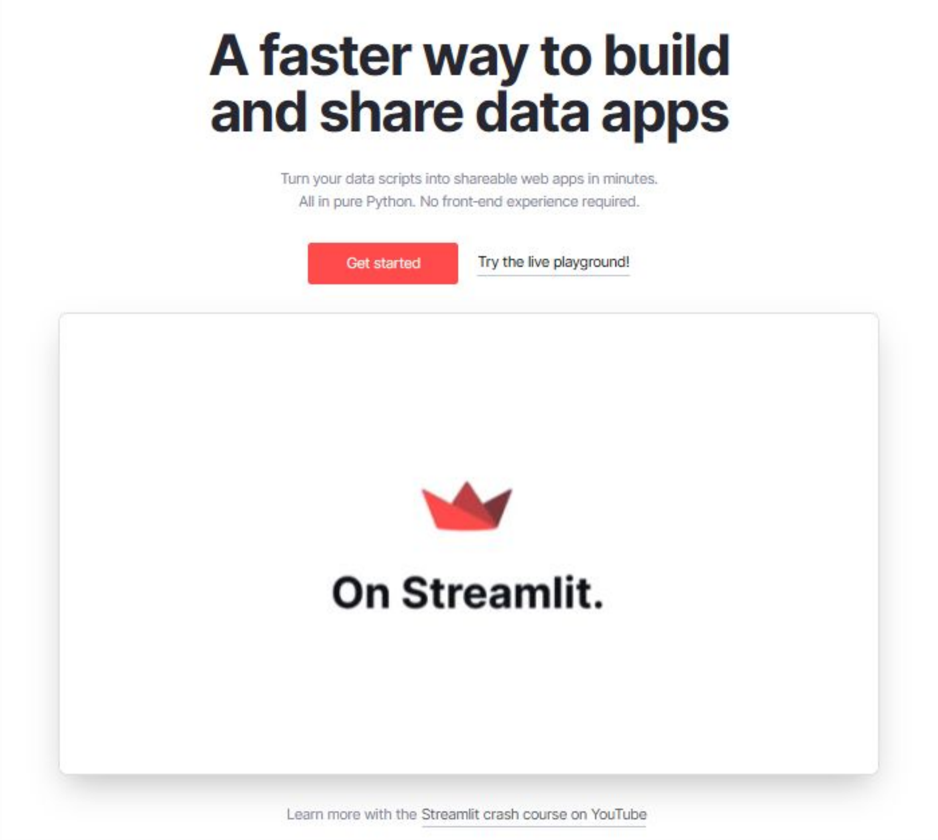
### Linux/MacOS

```
$ source .venv/bin/activate
```

# Virtual Environment - Deactivation

To exit the virtual environment, simply enter **`deactivate`** on any console

```
$ deactivate
```



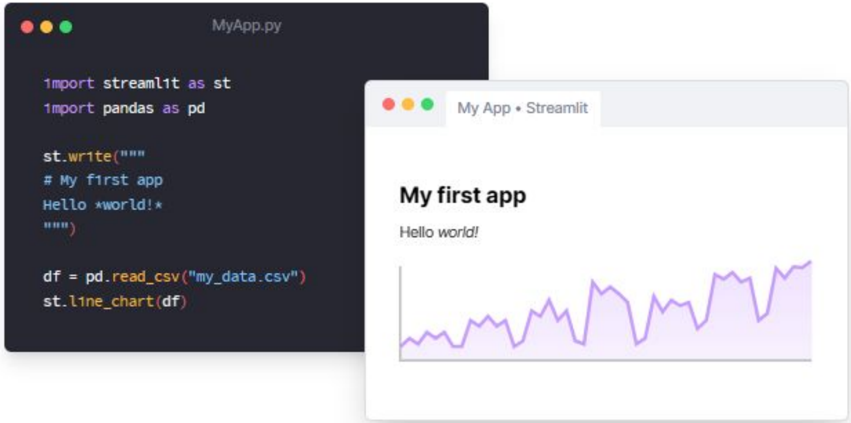
# A faster way to build and share data apps

Turn your data scripts into shareable web apps in minutes.
All in pure Python. No front-end experience required.

Learn more with the Streamlit crash course on YouTube

## Embrace scripting

Build an app in a few lines of code with our magically simple API. Then see it automatically update as you iteratively save the source file.



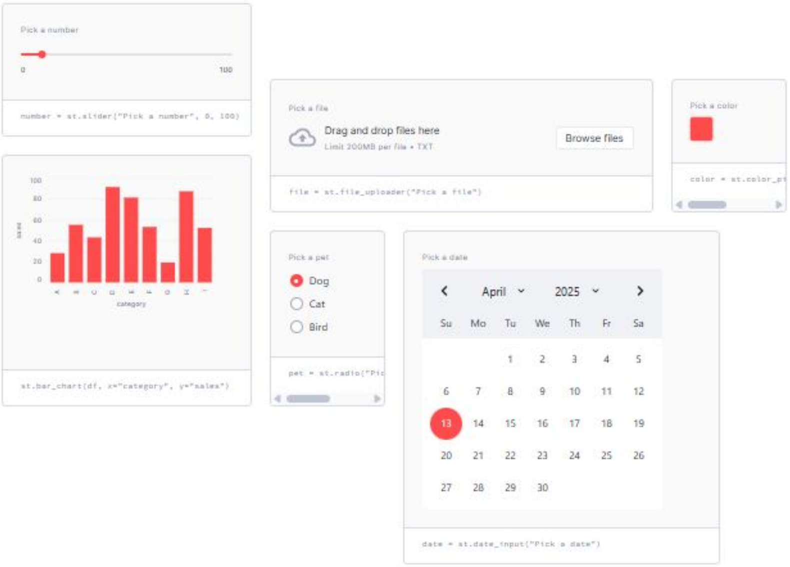
```
import streamlit as st
import pandas as pd

st.write("""
# My first app
Hello *world!*
""")

df = pd.read_csv("my_data.csv")
st.line_chart(df)
```

My first app

Hello *world!*

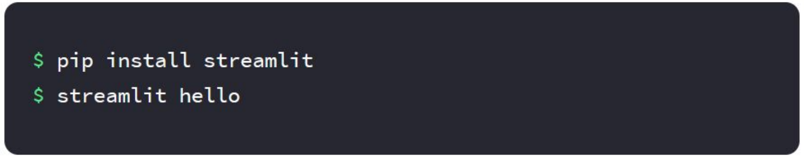
## Weave in interaction

Adding a widget is the same as declaring a variable. No need to write a backend, define routes, handle HTTP requests, connect a frontend, write HTML, CSS, JavaScript, ...

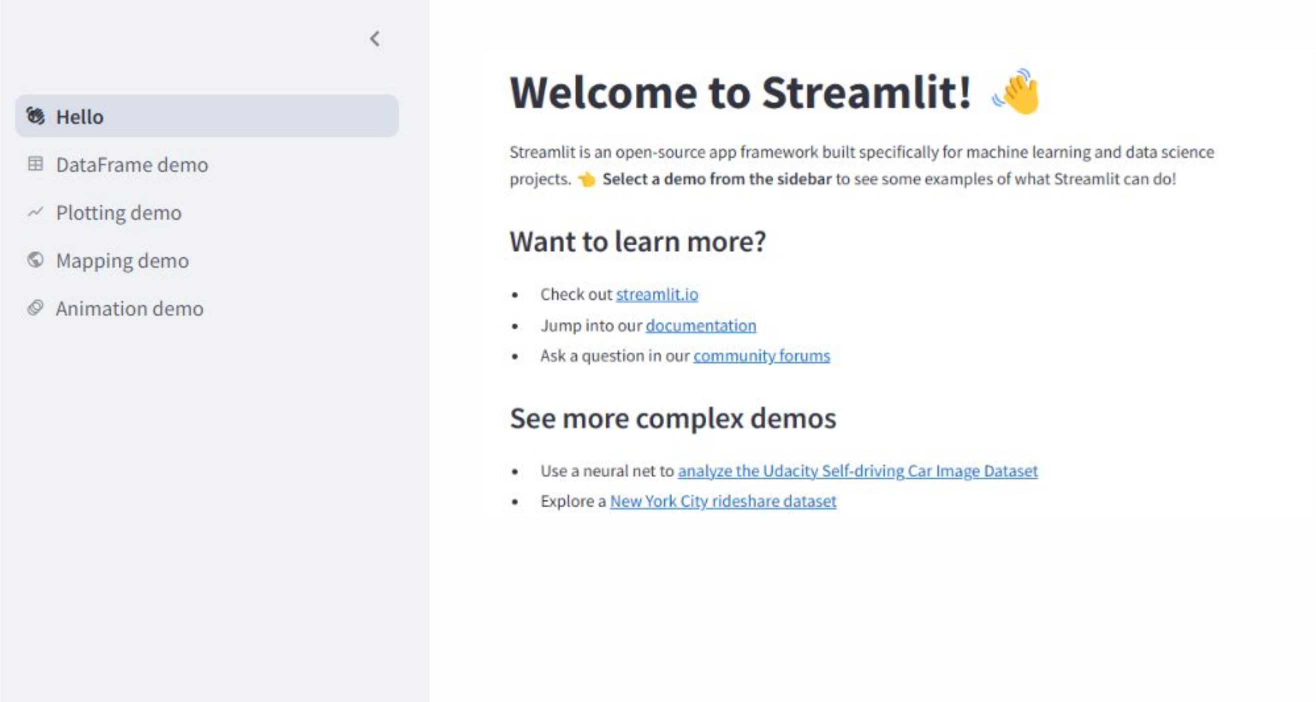


```
number = st.slider("Pick a number", 0, 100)
st.bar_chart(df, x="category", y="sales")
file = st.file_uploader("Pick a file")
color = st.color_pi
pet = st.radio("Pic
date = st.date_input("Pick a date")
```

# Get started in under a minute



```
$ pip install streamlit
$ streamlit hello
```



## Welcome to Streamlit! 👋

Streamlit is an open-source app framework built specifically for machine learning and data science projects. 👈 **Select a demo from the sidebar** to see some examples of what Streamlit can do!

### Want to learn more?

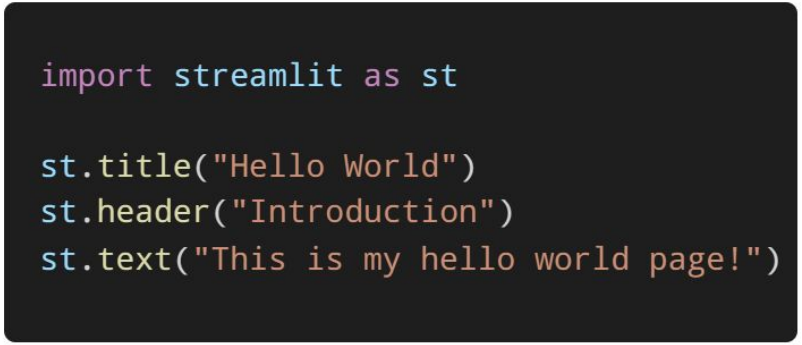
- Check out streamlit.io
- Jump into our documentation
- Ask a question in our community forums

### See more complex demos

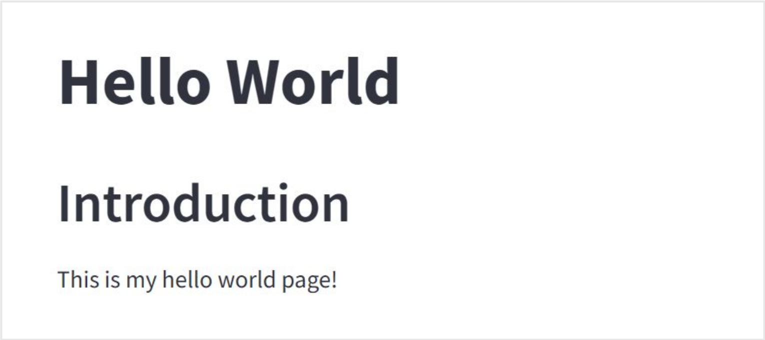
- Use a neural net to analyze the Udacity Self-driving Car Image Dataset
- Explore a New York City rideshare dataset

# Streamlit: Hello World

Make a new file with the following Python code.



```python
import streamlit as st

st.title("Hello World")
st.header("Introduction")
st.text("This is my hello world page!")
```



# Hello World

## Introduction

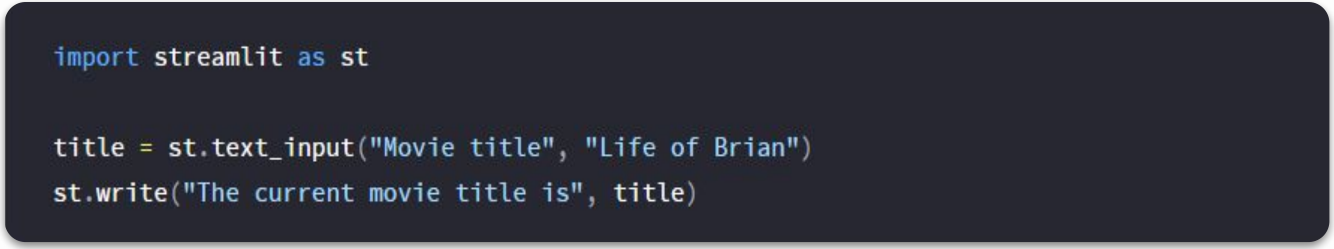
This is my hello world page!

# Components

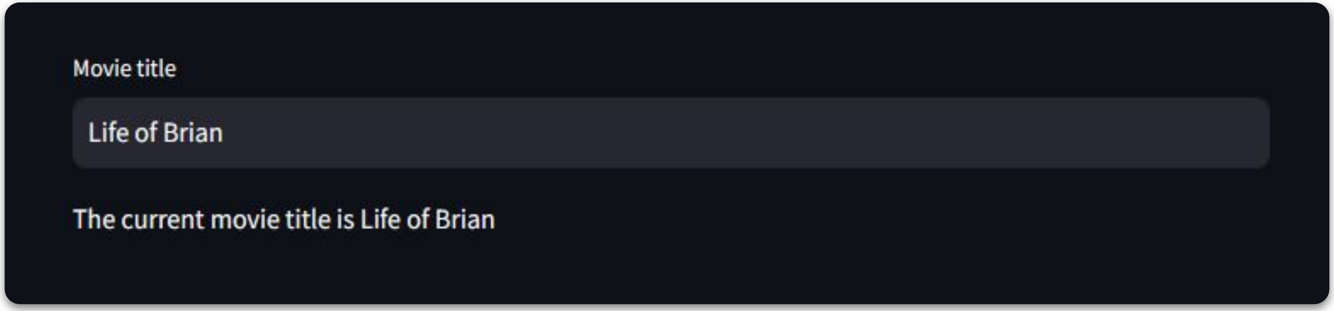
Learn some of the available interactive elements

# Text Input

The **`st.text_input`** displays a single-line text input widget.



```
import streamlit as st

title = st.text_input("Movie title", "Life of Brian")
st.write("The current movie title is", title)
```



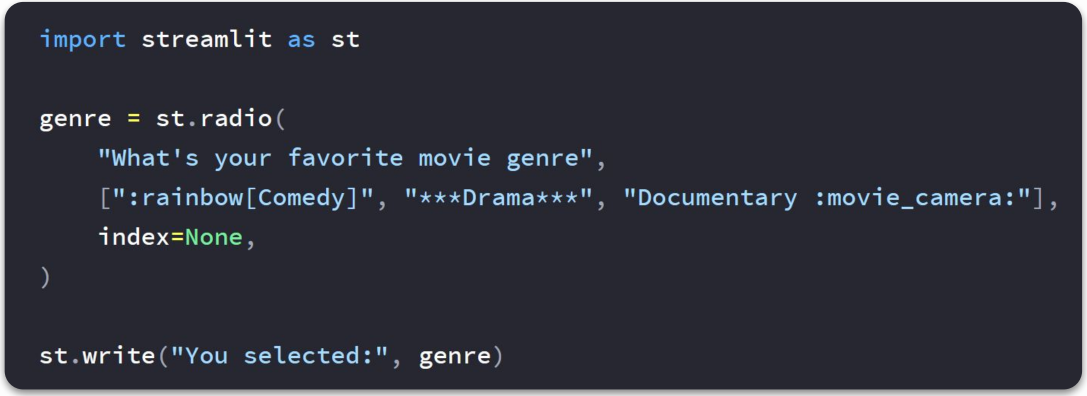
Movie title

Life of Brian

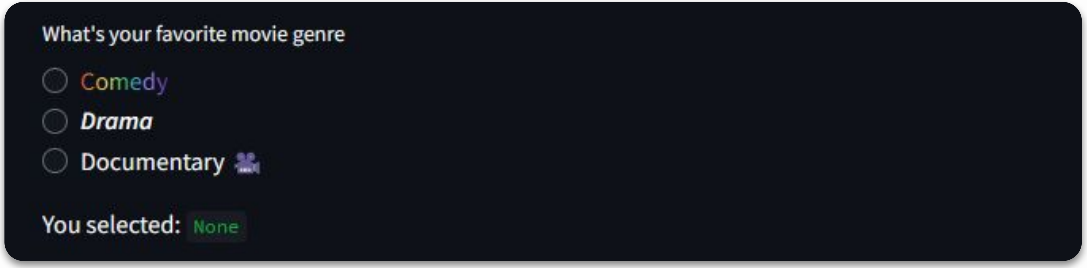
The current movie title is Life of Brian

# Radio Buttons

The **`st.radio`** displays a radio button widget



```
import streamlit as st

genre = st.radio(
    "What's your favorite movie genre",
    [":rainbow[Comedy]", "***Drama***", "Documentary :movie_camera:"],
    index=None,
)

st.write("You selected:", genre)
```



What's your favorite movie genre

○ Comedy

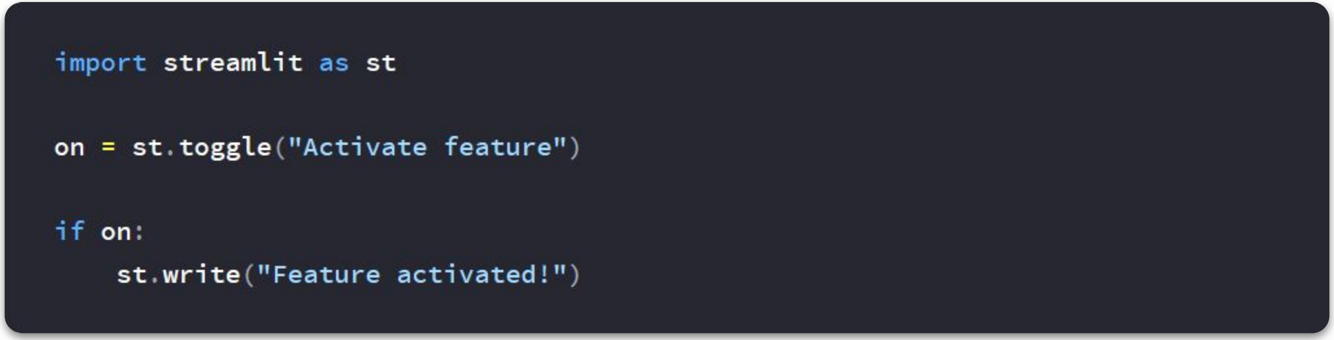
○ ***Drama***

○ Documentary 🎥

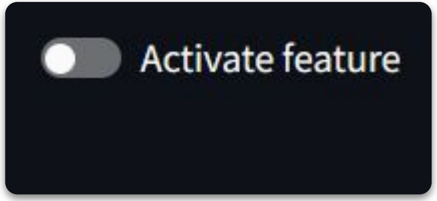
You selected: `None`

# Toggle

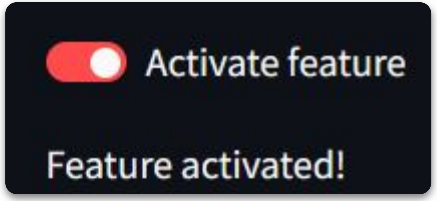
The **`st.toggle`** displays a slider widget for integers, time, and datetime values



```python
import streamlit as st

on = st.toggle("Activate feature")

if on:
    st.write("Feature activated!")
```



Activate feature

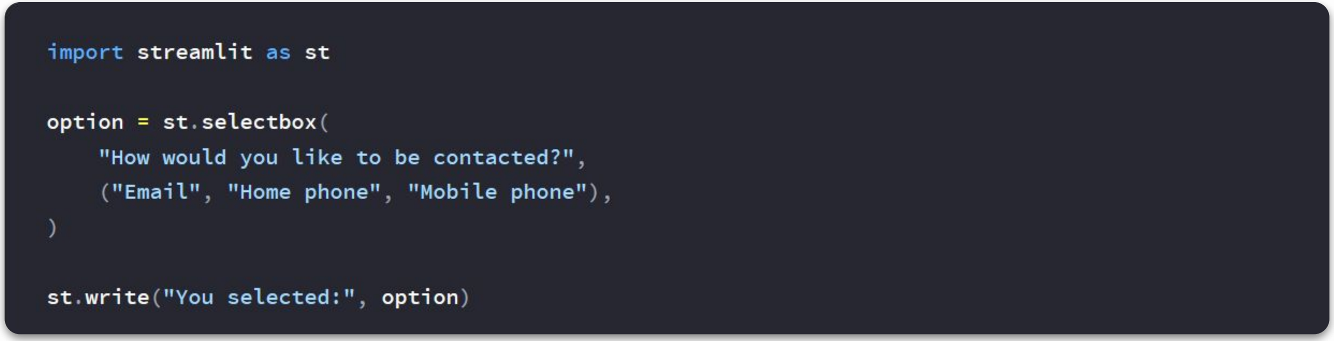


Activate feature

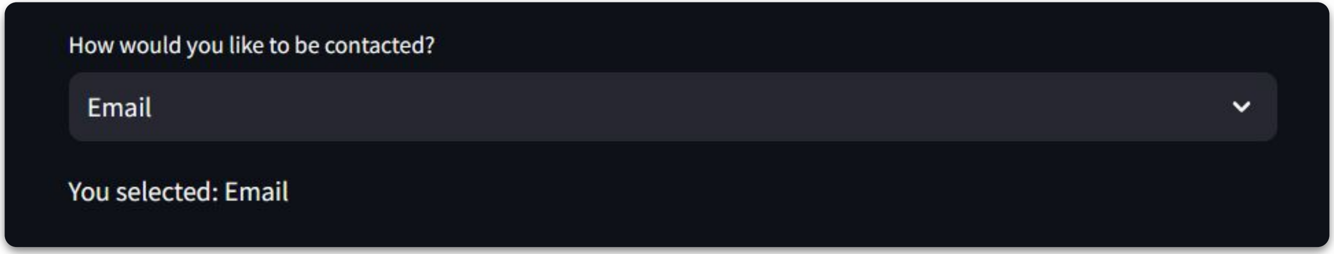
Feature activated!

# Select Box

The **`st.select_box`** displays a select widget for choosing a single value



```
import streamlit as st

option = st.selectbox(
    "How would you like to be contacted?",
    ("Email", "Home phone", "Mobile phone"),
)

st.write("You selected:", option)
```



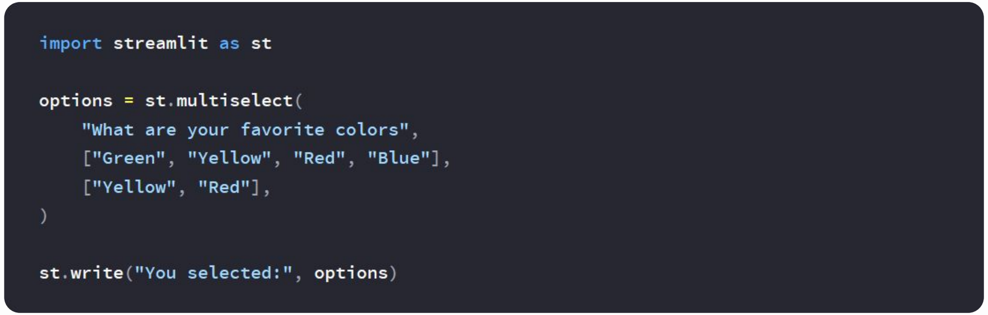
How would you like to be contacted?

Email

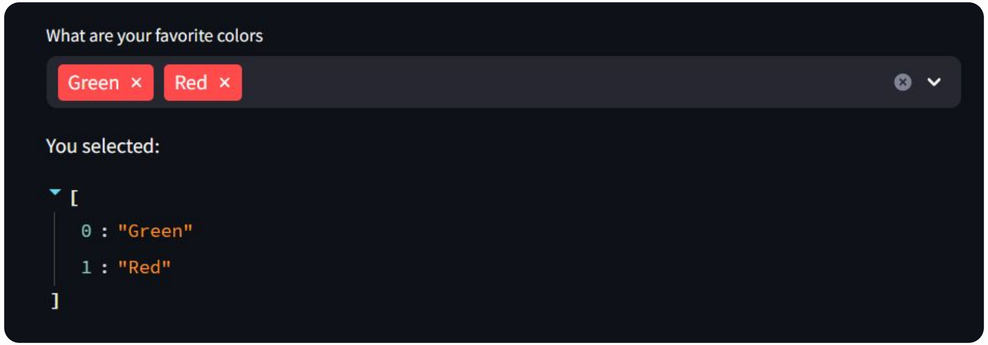
You selected: Email

# Multiselect

The **`st.multiselect`** displays a multiselect widget



```
import streamlit as st

options = st.multiselect(
    "What are your favorite colors",
    ["Green", "Yellow", "Red", "Blue"],
    ["Yellow", "Red"],
)

st.write("You selected:", options)
```



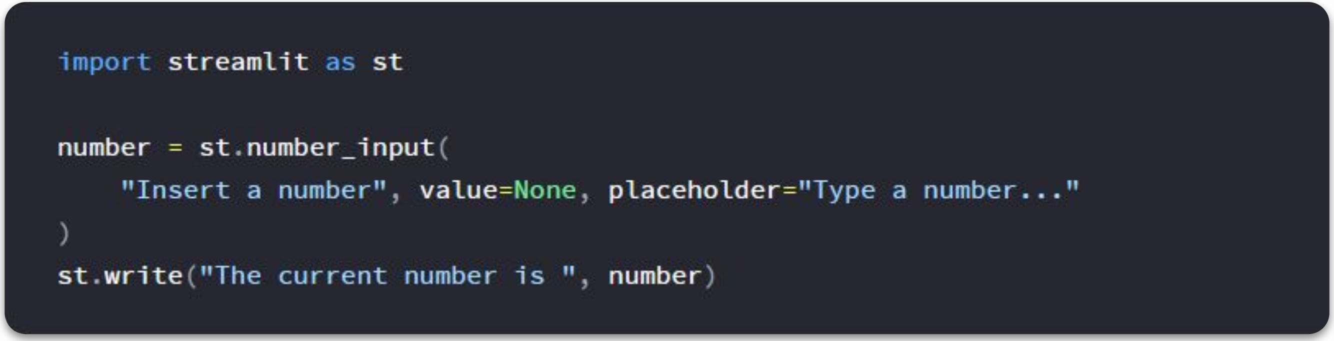
What are your favorite colors

You selected:

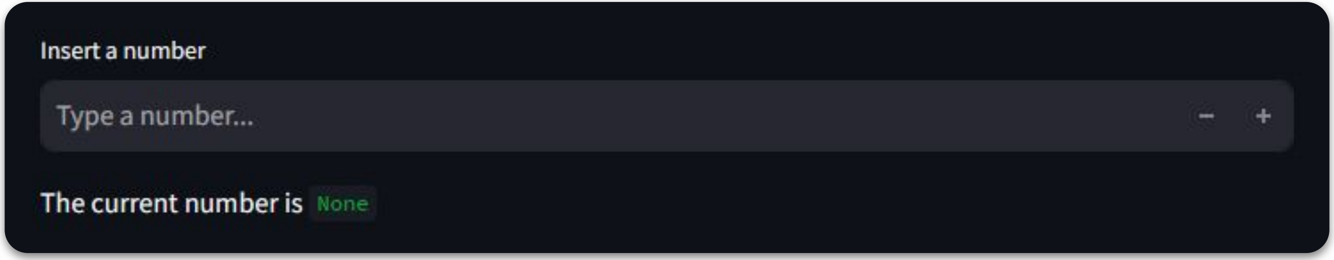
```
[
  0 : "Green"
  1 : "Red"
]
```

# Number Input

The **`st.number_input`** displays a numeric input widget



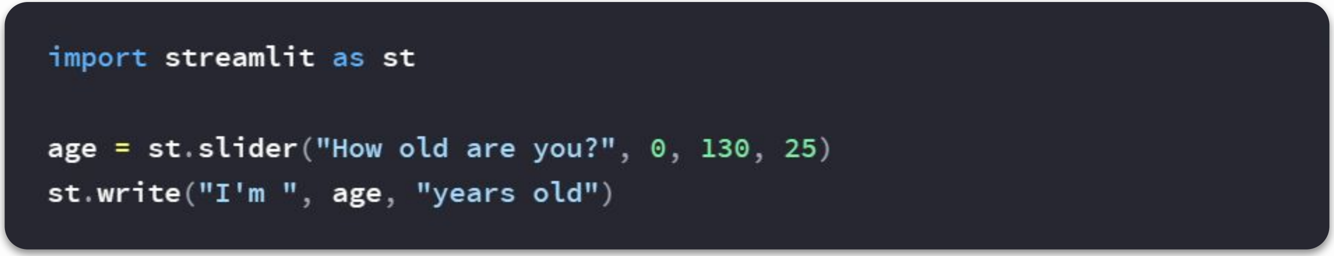
```python
import streamlit as st

number = st.number_input(
    "Insert a number", value=None, placeholder="Type a number..."
)
st.write("The current number is ", number)
```



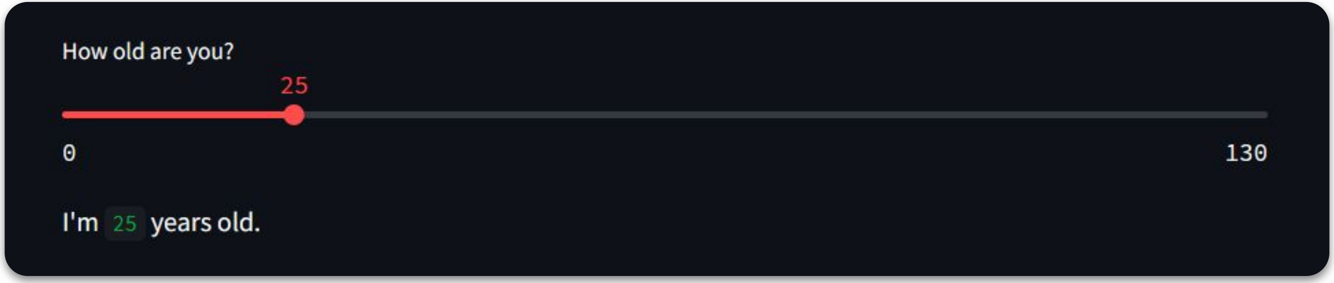
The current number is `None`

# Slider

The **`st.slider`** displays a slider widget for integers, time, and datetime values



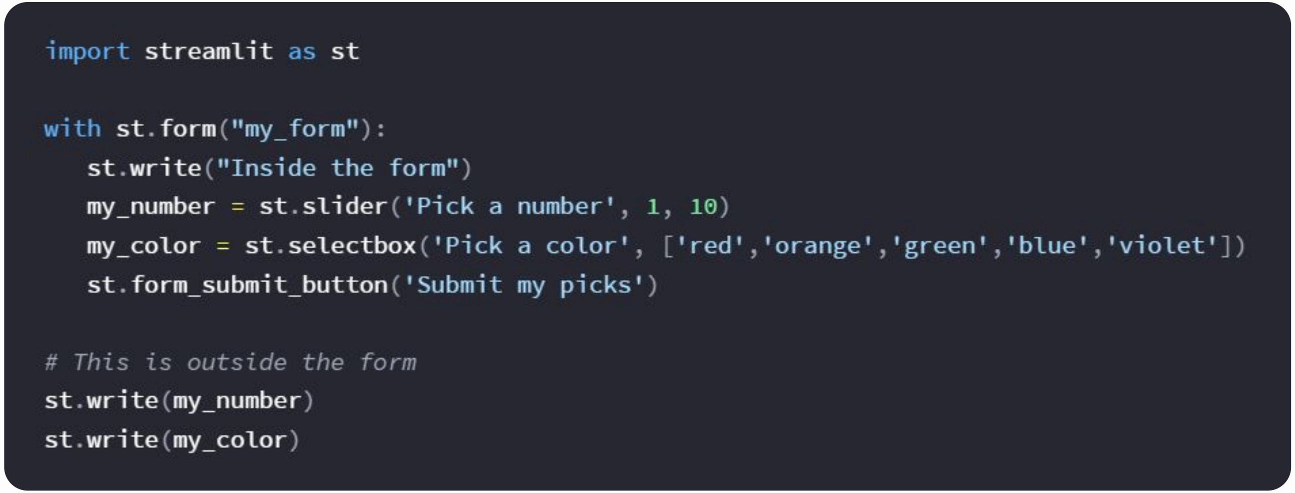
```python
import streamlit as st

age = st.slider("How old are you?", 0, 130, 25)
st.write("I'm ", age, "years old")
```



I'm 25 years old.

# Submit Form

The **`st.form`** ensures that every input change doesn’t refresh the page every time



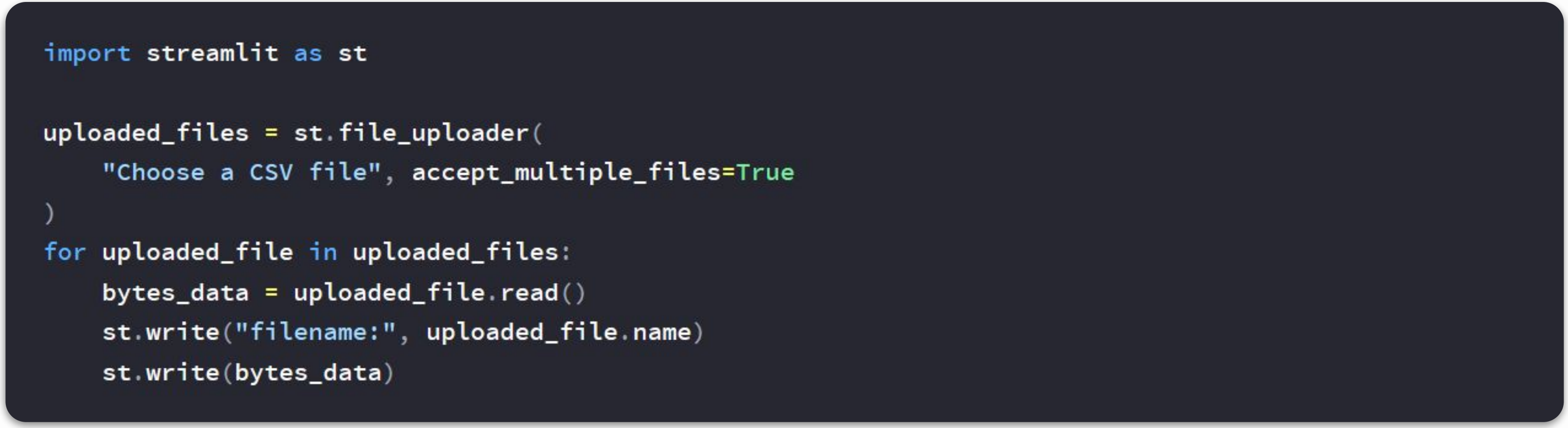
```python
import streamlit as st

with st.form("my_form"):
   st.write("Inside the form")
   my_number = st.slider('Pick a number', 1, 10)
   my_color = st.selectbox('Pick a color', ['red','orange','green','blue','violet'])
   st.form_submit_button('Submit my picks')

# This is outside the form
st.write(my_number)
st.write(my_color)
```

# Data Handling

Process and visualize more data-intensive processes

# Upload Files

Run the following on your chosen terminal to setup commits and remote connections



```python
import streamlit as st

uploaded_files = st.file_uploader(
    "Choose a CSV file", accept_multiple_files=True
)
for uploaded_file in uploaded_files:
    bytes_data = uploaded_file.read()
    st.write("filename:", uploaded_file.name)
    st.write(bytes_data)
```

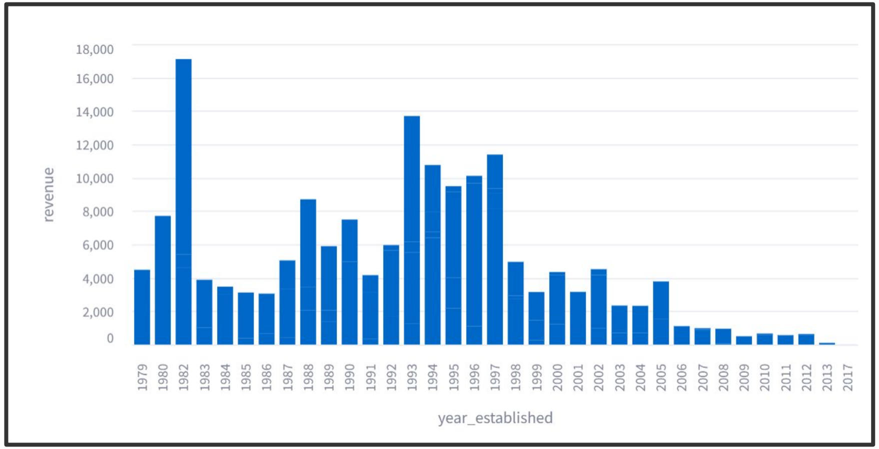
# Read CSV and Excel File

Run the following on your chosen terminal to setup commits and remote connections

```
import streamlit as st
import pandas as pd

uploaded_file = st.file_uploader("File:", type=["csv", "xlsx", "xls"])

if uploaded_file is not None:
    st.write(f"Uploaded file: {uploaded_file.name}")

    if uploaded_file.name.endswith(".csv"):
        df = pd.read_csv(uploaded_file)
    elif uploaded_file.name.endswith((".xlsx", ".xls")):
        df = pd.read_excel(uploaded_file)

    st.write(df)
```

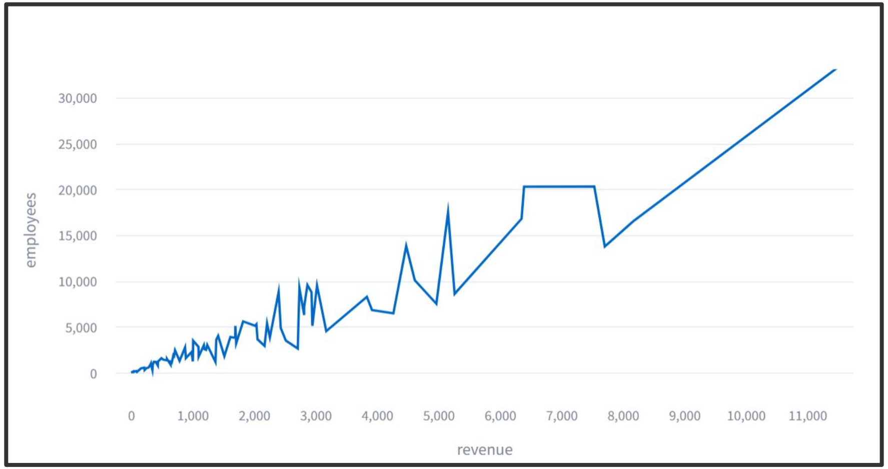
# Bar Chart

```
import streamlit as st
import pandas as pd

df = pd.read_csv("data/sales/accounts.csv")
st.bar_chart(df, x="year_established", y="revenue")
```



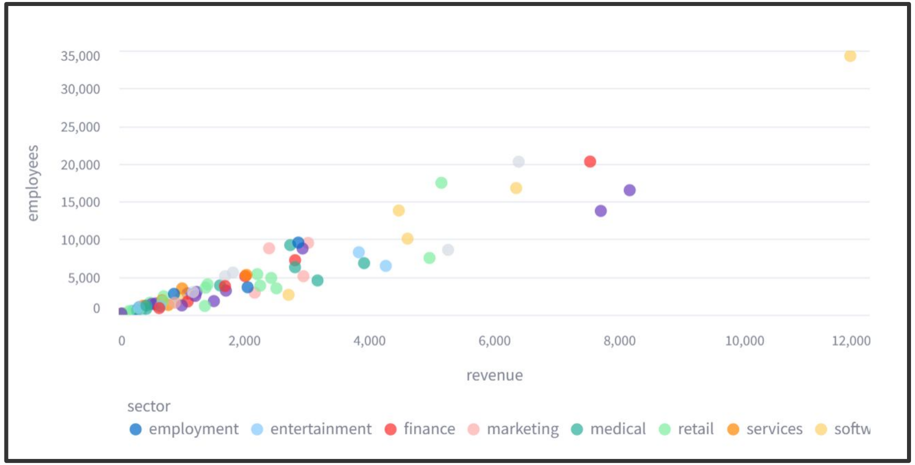
# Line Plot

```python
import streamlit as st
import pandas as pd

df = pd.read_csv("data/sales/accounts.csv")
st.line_chart(df, x="revenue", y="employees")
```



# Scatter Chart

```
import streamlit as st
import pandas as pd

df = pd.read_csv("data/sales/accounts.csv")
st.scatter_chart(df, x="revenue", y="employees", color="sector")
```

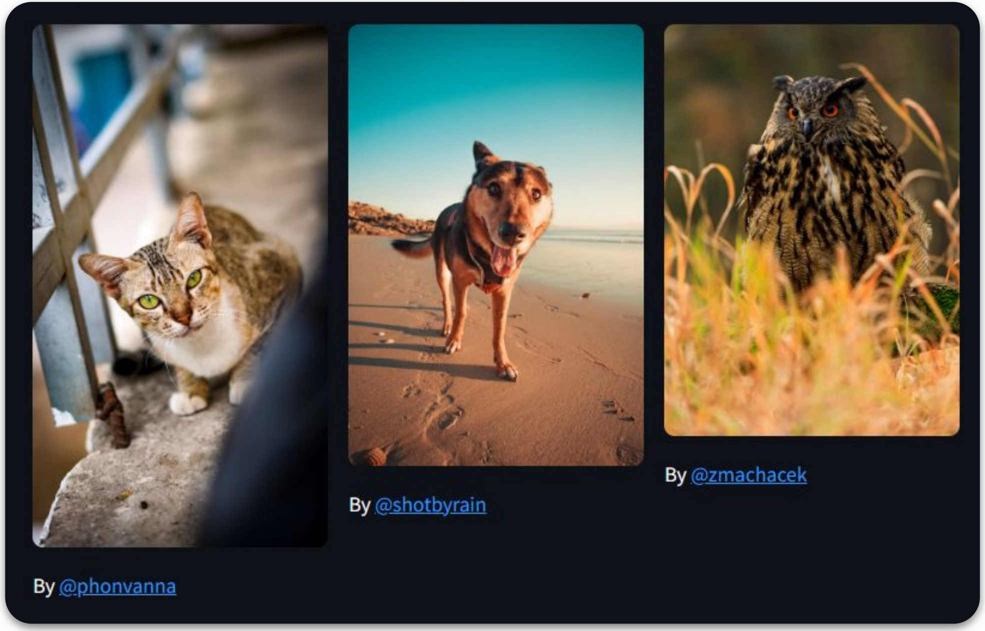


# Modularization

High-level Streamlit code organization

# Column Layouting

Streamlit supports multi-column layouts



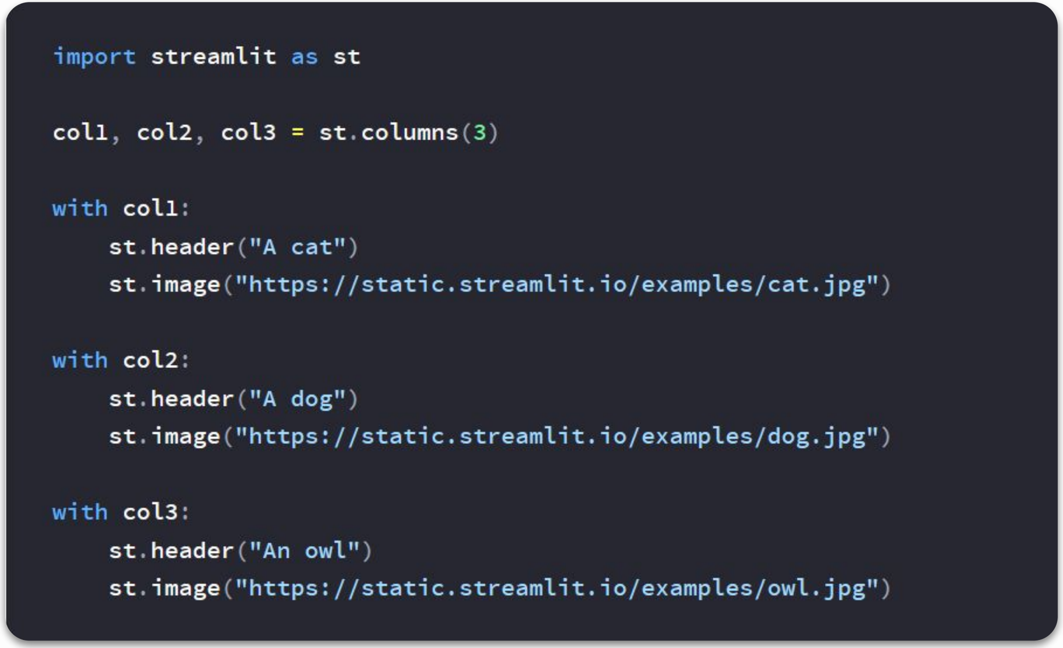
By @phonvanna

By @shotbyrain

By @zmachacek

# Columns

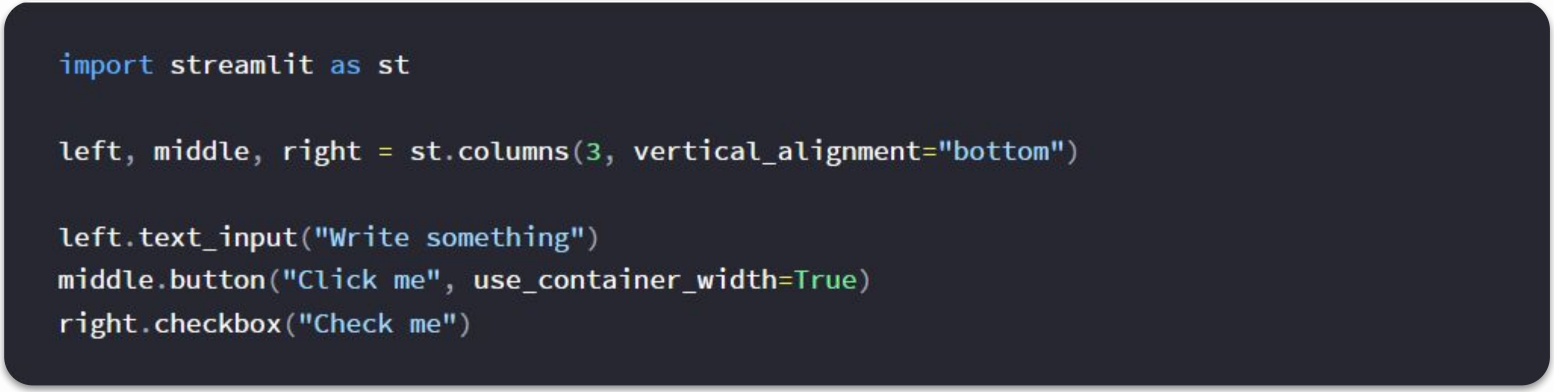
Using the context handler **`with`** syntax, content will be divided into separate columns



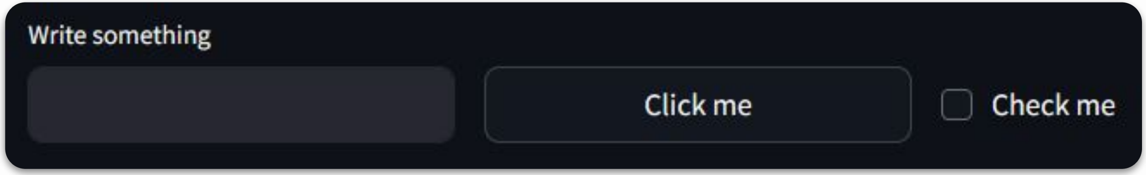
```python
import streamlit as st

col1, col2, col3 = st.columns(3)

with col1:
    st.header("A cat")
    st.image("https://static.streamlit.io/examples/cat.jpg")

with col2:
    st.header("A dog")
    st.image("https://static.streamlit.io/examples/dog.jpg")

with col3:
    st.header("An owl")
    st.image("https://static.streamlit.io/examples/owl.jpg")
```

# Simple Column Layout

For simple columns, **`st`** can be replaced with the given column name

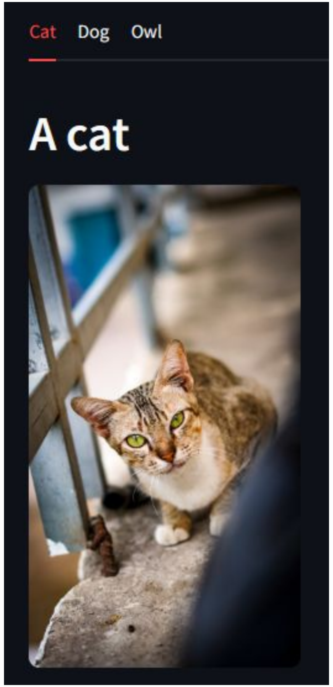


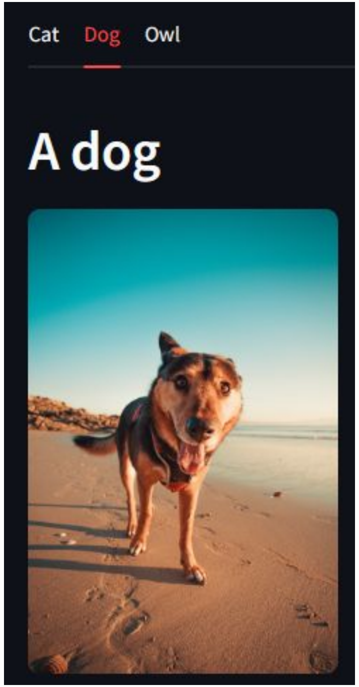
```python
import streamlit as st

left, middle, right = st.columns(3, vertical_alignment="bottom")

left.text_input("Write something")
middle.button("Click me", use_container_width=True)
right.checkbox("Check me")
```

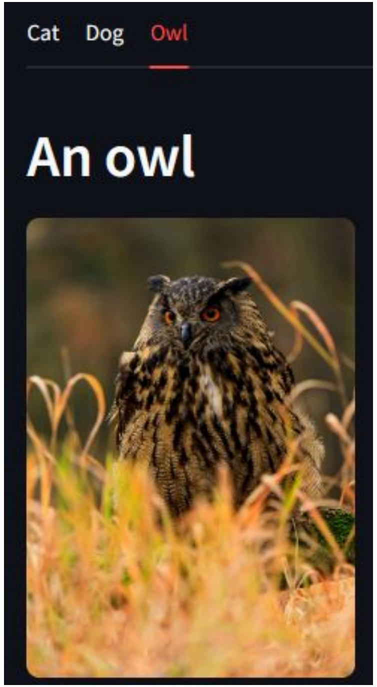


# Tabs

Streamlit also supports tab layouts to prevent cluttering the page

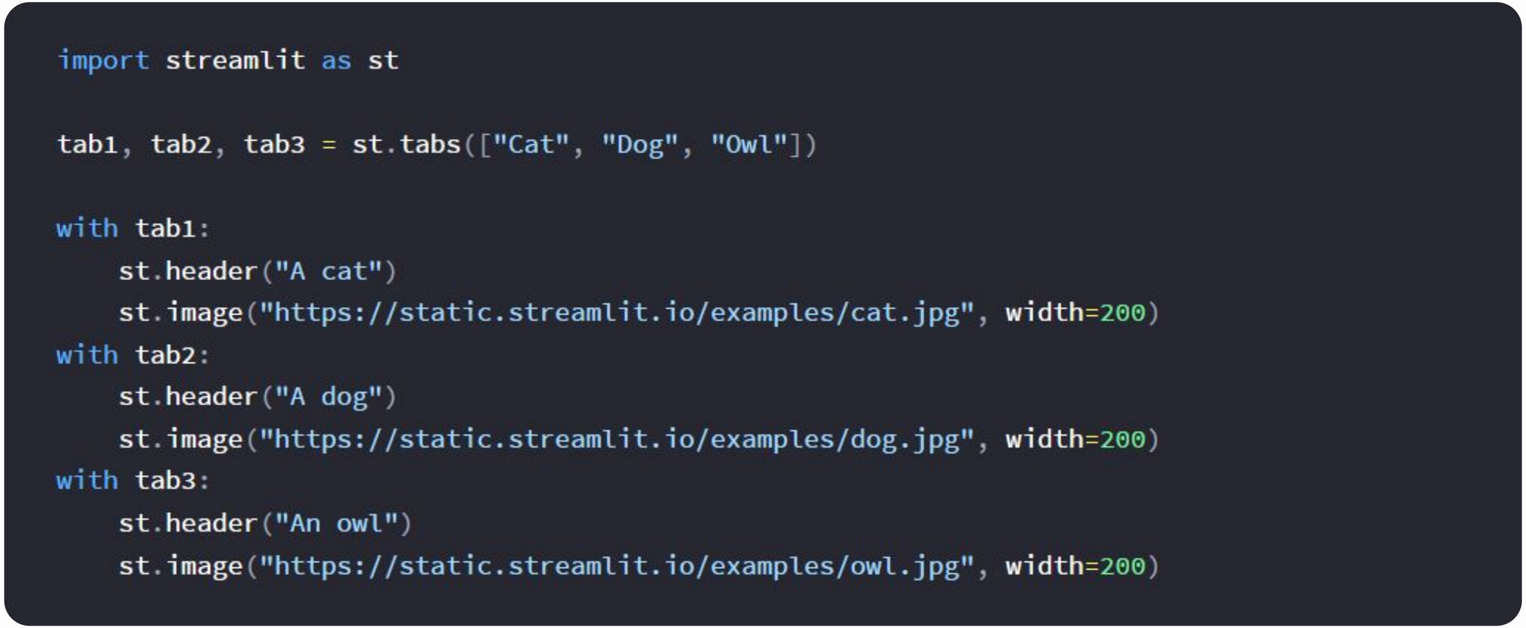






# Tabs

Using the context handler **`with`** syntax, content will be divided into separate tabs



```python
import streamlit as st

tab1, tab2, tab3 = st.tabs(["Cat", "Dog", "Owl"])

with tab1:
    st.header("A cat")
    st.image("https://static.streamlit.io/examples/cat.jpg", width=200)
with tab2:
    st.header("A dog")
    st.image("https://static.streamlit.io/examples/dog.jpg", width=200)
with tab3:
    st.header("An owl")
    st.image("https://static.streamlit.io/examples/owl.jpg", width=200)
```

# Multiple Pages

Multiple subpages are easy to implement in Streamlit. Place subpages in the **pages/** folder

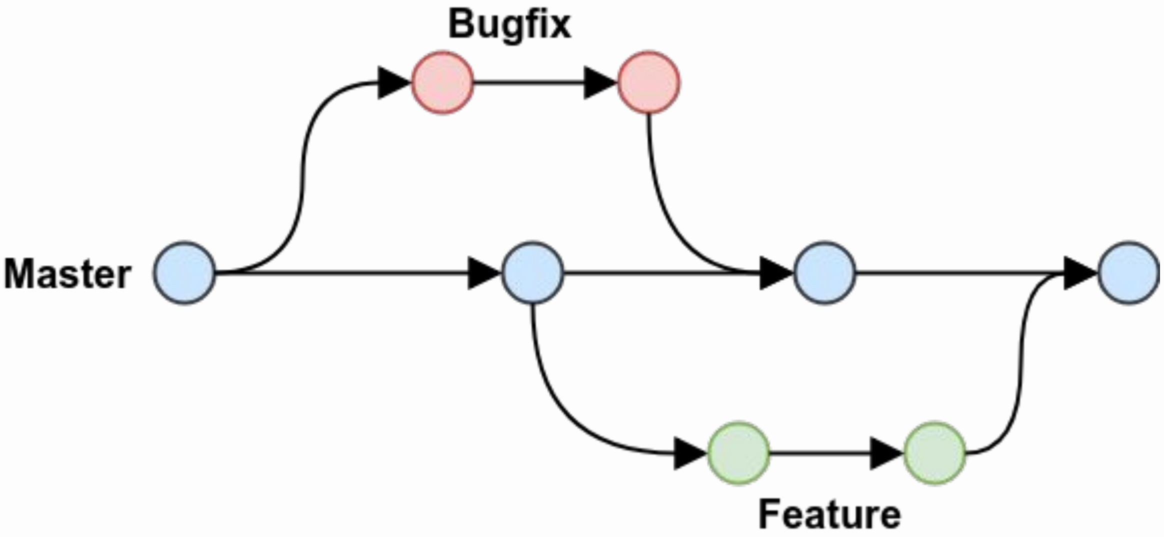
```
.
└── project_name/
    ├── ...
    └── src/
        ├── pages/
        │   ├── subpage1.py
        │   ├── subpage2.py
        │   └── subpage3.py
        └── main.py
```

# Version Control

Taught in the context of git

# Git

**`Git`** is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.



# Git Project Setup

Run the following on your chosen terminal to setup commits and remote connections

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your@email.com"
```

For every new project, open the project terminal in the terminal and run this

```
$ git init
```

# Git Clone

To create a local copy of an online repository, run this command. This doesn’t need **`git init`**

```
$ git clone source
```

Here is an example of an existing repository from Github

```
$ git clone https://github.com/Ayumu098/quotes.git
```

# Git Create Branch

To see the list of existing branches, run the following command

```
$ git branch
```

To create a new branch in your repository, run the following command

```
$ git switch -c feature/my-feature
```

# Git Stage

To save changes in your local repository,  you need to stage or note what files to track.

```
$ git add filename1.py
$ git add filename2.py
$ ...
```

You can determine what files have been modified from last time with this command

```
$ git status
```

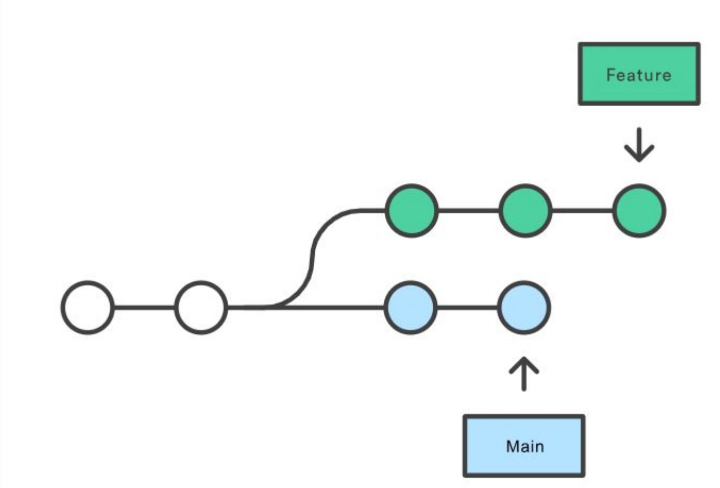
You can also stage all of the changes using this command

```
$ git add .
```

# Git Commit

After staging the changes, the last step to saving the changes locally is to commit.

```
$ git commit -m "Describe changes (Verb - Subject - Details)"
```



# Git Pull

To ensure the current branch is in sync with the online repository, run the following

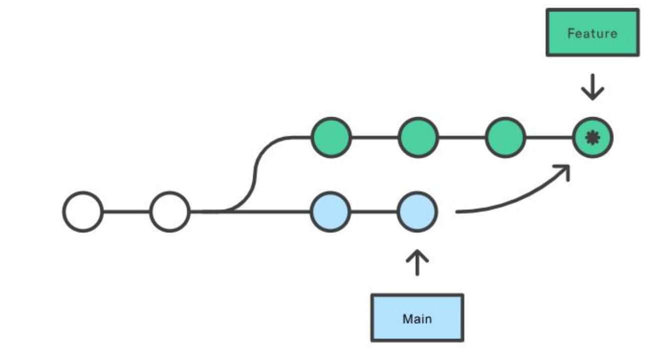
```
$ git switch main
$ git pull --rebase origin main
$
$ git switch feature/my-feature
$ git pull --rebase origin main
```

# Git Push and Pull Requests

Finally, reflect the changes in the feature branch to the online repository with this command

```
$ git push origin feature/my-feature
```

To merge the changes in the feature with the develop or main branch, make a pull request on your chosen online repository platform. It can be done in console but this is better for code reviews and tests.



# Beautiful Soup

Getting data from the web

# Parsing a string

Beautiful soup can handle string files directly

```python
from bs4 import BeautifulSoup

soup = BeautifulSoup("<html>a web page</html>", 'html.parser')
```

# Parsing a document

Beautiful soup can also parse or open html files

```python
from bs4 import BeautifulSoup

with open("index.html") as file:
    soup = BeautifulSoup(file, 'html.parser')
```

# Parsing a website online

Using the requests library, beautiful soup can also directly parse live websites

```python
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
```

# Tags

Every detected component in the parser is a **Tag** object

```python
from bs4 import BeautifulSoup

soup = BeautifulSoup(
    '<b class="boldest">Extremely bold</b>',
    'html.parser'
)
tag = soup.b
print(type(tag))
```

# Tag Name

To access the HTML tag of the object, use the **name** field

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(
    '<b class="boldest">Extremely bold</b>',
    'html.parser'
)
tag = soup.b
print(tag.name)
```

# Tag String

To access the string contents the object, use the **`string`** field

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(
     '<b class="boldest">Extremely bold</b>',
     'html.parser'
)
tag = soup.b
print(tag.string)
```

# Finding tags (Explicit)

Soup and Tag objects have a find method that can be used to search for HTML tags.

```python
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
    print(soup.find("head"))
```

# Finding tags (Implicit)

Soup and Tag objects can also find tags using attribute access. It returns None if not found.

```python
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
    print(soup.head)
```

# Finding tags (Nested)

Tag finding can be nested using attribute access

```
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
    print(soup.body.h1)
```

# Multiple Finding

To check for a tag in a soup or existing tag, use the `findAll` method

```
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
    print(soup.findAll('a'))
```

# Multiple Finding (Chained Conditions)

The `findAll` method can also accept additional inputs to narrow down a search

```python
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
     print(soup.findAll('a','head'))
```

# Finding using ID’s

Finally, the method can also find components using their id

```python
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
    print(soup.findAll(id='a'))
```

# Finding using Classes

Additionally, the findAll method can also find components using their id

```
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
    print(soup.findAll("p", class_="body"))
```

# Finding using Strings

Finally, the `findAll` method can also find tags containing specific strings directly

```python
from bs4 import BeautifulSoup
import requests

url = "https://www.bbc.com/news"
response = requests.get(url)

if response.status_code == 200:
    soup = BeautifulSoup(response.text, "html.parser")
    print(css_soup.find_all("p", string="Elsie"))
```

05

# Lab Session

# Additional References

Additional references you can look into:

**Books**

- Automate the Boring Stuff with Python
- Python Distilled
- Fluent Python

**YouTube**

- CS50 - CS50P Python
- Bro Code - Python Full Course
- Corey Schafer - Python Playlist

# Recommended Next Steps

For more intermediate development, read on the following topics

**External Libraries**

- Web Scraping: Beautiful Soup, Requests, Scrapy
- Web Development: Django, FastAPI
- Data Science: Sklearn, Pandas, Seaborn

**Internal Libraries**

- Refactoring: functools, Itertools, contextlib
- File Management: pathlib, shutil, os, tempfile

# pass: happycoding

stephen.singer.098@gmail.com

# Python: Day 04

Advanced Programming