Introduction to Programming with Python

Dr. Anatol Wegner

Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany

anatol.wegner@uni-wuerzburg.de

Lecture 11
Repetitorium

January 31, 2025

Big Recap

- **Python Basics:**
  - *Lecture 1:* Setting up Python and 'Hello world!'
  - *Lecture 2:* Data and control structures
  - *Lecture 3:* Functions and classes
  - *Lecture 4:* Modules and file handling basics

Big Recap

- **Data analysis and machine learning:**
  - *Lecture 5:* Pandas and Seaborn
  - *Lecture 6:* Machine learning basics with sklearn.
  - *Lecture 7:* Natural language processing with nltk

Big Recap

- **Other key concepts and tools:**
  - *Lecture 8:* Version control with git
  - *Lecture 9:* Database basics with sqlite
  - *Lecture 10:* Parallel processing with multiprocessing

Lecture 1: Setting up Python and 'Hello world!'

- Basics of programming
- Compiled vs interpreted programming languages
- Overview of the Python programming language
- Setting up Python and IDE
- We wrote our first program in Python
- We learned about **basic variable types and operations**

What is programming?

1. the process creating an instructional program for a device to make it perform a certain task.
2. attempting to get a computer to complete a specific task without making mistakes.
3. the process of creating precise set instructions for a computer to perform a well defined task without mistakes.

- modern computers work with (binary) machine-level code to encode data and instructions
- higher level programing languages allow us to writte programs that are closer to human language (usually English)

Macbook computer

image credit: Wikipedia, VladIsLoveW, CC-BY-SA

Programming languages

- just as natural language, programing languages have rules and conventions, synthax and semantics
  - a set of **precise/formal** rules (grammar) that define what is valid code and/not
- compilers and interpreters are programs that translate higher level programming languages to machine-code

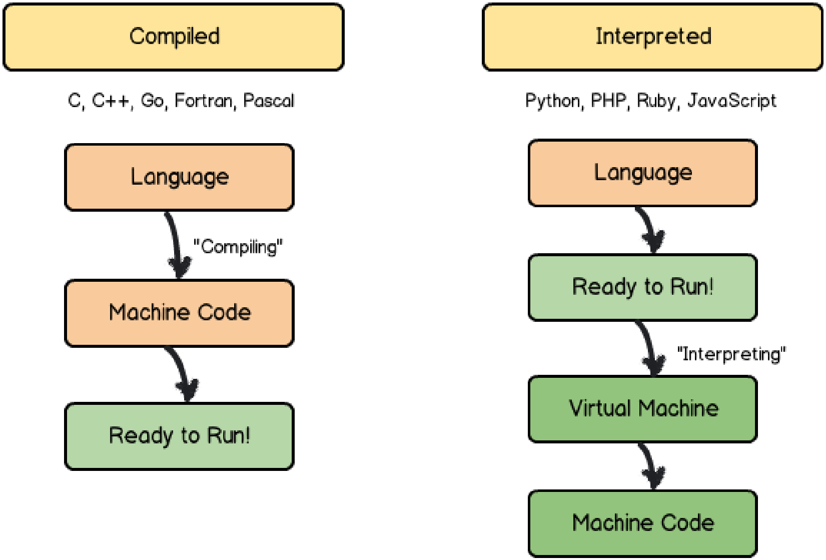
Some popular computer languages

Programming languages

- just as natural language, programing languages have rules and conventions, synthax and semantics
  - a set of **precise/formal** rules (grammar) that define what is valid code and/not
- compilers and interpreters are programs that translate higher level programming languages to machine-code
- there are many programming languages: choices of programming language depends on personal preference and application.

Some popular computer languages

Compiled vs Interpreted languages



Compiled vs. Interpreted Languages

- **compiler translates program** in high-level language to machine code **before** it can be executed
    - compiled binaries are not portable
    - users may need to compile source code
    - each change requires recompilation

Compiled vs. Interpreted Languages

- **compiler translates program** in high-level language to machine code **before** it can be executed
  - compiled binaries are not portable
  - users may need to compile source code
  - each change requires recompilation
- **interpreter directly executes instructions** in high-level programming language

Definition

An **interpreter** is a program that directly executes instructions written in a programming language, without requiring its prior compilation to machine code.

Compiled vs. Interpreted Languages

- **compiler translates program** in high-level language to machine code **before** it can be executed
    - compiled binaries are not portable
    - users may need to compile source code
    - each change requires recompilation
- **interpreter directly executes instructions** in high-level programming language
- no need for (re)compilation, no non-portable binaries

Definition

An **interpreter** is a program that directly executes instructions written in a programming language, without requiring its prior compilation to machine code.

Compiled vs. Interpreted Languages

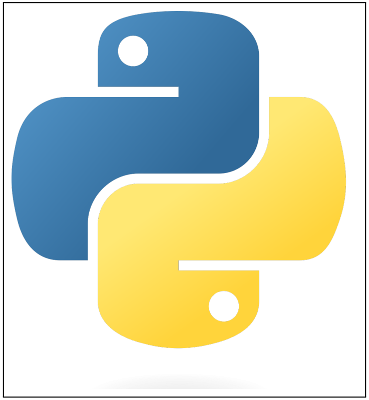
- **compiler translates program** in high-level language to machine code **before** it can be executed
  - compiled binaries are not portable
  - users may need to compile source code
  - each change requires recompilation
- **interpreter directly executes instructions** in high-level programming language
- no need for (re)compilation, no non-portable binaries
- interpreted languages are **typically slower than compiled languages** (but not necessarily)

Definition

An **interpreter** is a program that directly executes instructions written in a programming language, without requiring its prior compilation to machine code.

Python

- `python` is the most **popular interpreted programming language**
- widely-used for **data processing, analytics, and machine learning**
- object-oriented, dynamically typed, automatic memory management



Python

- python is the most **popular interpreted programming language**
- widely-used for **data processing, analytics, and machine learning**
- object-oriented, dynamically typed, automatic memory management
- user-friendly, great for **beginners in programming**
- **rich ecosystem of libraries** (modules) that implement almost any imaginable functionality

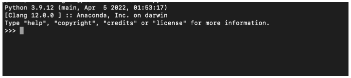
Guido van Rossum, developer of python

image credit: Wikpedia, Doc Searls, CC BY-SA 2.0

Interacting with Python

Interactive Python shell:

- open terminal and type 'Python'
- can execute Python commands
- code is executed *sequentially*



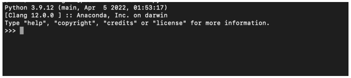
```
Python 3.9.12 (main, Apr  5 2022, 01:53:17)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The Python shell

Interacting with Python

Execute a Python (.py) file:

1. write code using any text editor,
2. save with extension .py (e.g Hello.py)
3. run using »Python Hello.py
4. will run the code line by line.



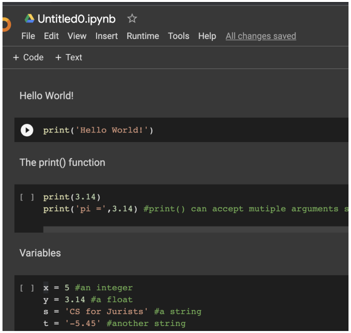
```
Python 3.9.12 (main, Apr  5 2022, 01:53:17)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The Python shell

Interacting with Python

Jupyter notebooks:

- interactively edit & execute code
- code is executed in blocks
- display visualizations (e.g graphs, images...)
- text blocks in MarkDown.
- we will mostly use Jupyter notebooks...
- see https://colab.google/ for a free online Jupyter server (requires google account)



Hello World!

```
print('Hello World!')
```

The print() function

```
print(3.14)
print('pi =',3.14) #print() can accept mutiple arguments s
```

Variables

```
x = 5 #an integer
y = 3.14 #a float
s = 'CS for Jurists' #a string
t = '-5.45' #another string
```

A jupyter notebook

Strings

- In Python text values are called strings
  - strings are just lists of characters
  - strings are defined using quotation marks (single or double)
- Examples:
  - a='Hello world' (a string)
  - b='3.14' (another string)
  - c=3.14 (not a string)

Basic operations

- Arithmetic operations (+, -, *, /, **, %):
  - x**y (x^y): 3**2 = 9
  - % (mod) : 3 % 2 = 1
- Comparisons :
  - 3==4 (False)
  - 3!=4 (True)
  - 3<4 (True)
  - 3>4 (False)
  - 3<=4 (True)
- Logical operations (and, or, not):
  - False and True (False)
  - False or True (True)
  - False or not True (False)

Warning 1

Not all operations work with all types

Warning 2

Output of operations depend on types

- ’Alan’+’Turing’=’AlanTuring’
- 3 + True = 4
- True + True = 2
- ’Alan’+3=? (error)

Lecture 2: Data and control structures

- we learned about **basic variable types and operations**
- we learned about basic Python objects such as **arrays, dictionaries and sets**

Lecture 2: Data and control structures

- we learned about **basic variable types and operations**
- we learned about basic Python objects such as **arrays, dictionaries and sets**
- we introduced **if, elif and else** statements
- we learned how to use **for** and **while** loops

Variables and basic types

- Variables are containers for data
  - numbers, text, lists, dictionaries, files...
- In Python variables are created at assignment:
  - x = 5, y = 3.14, a = ’CS for Jurists’
- Python is dynamically typed:
  - Python assigns types to variables depending on their current value
  - Types of variables can change over time: a=1, a=’John’
  - need to keep track of variable types.

Basic types

- int() : 0, 1, -2, 1982 etc.
- float(): 3.14, -4.62131 etc.
- str(): ’Apples’, ’John’ etc.
- bool(): True, False.

Type conversion

- int(3.14) = 3
- float(3) = 3.0
- str(-4.45) = ’-4.45’
- bool(3.14) = True
- bool(0.0) = False
- int(’apple’)=?

Basic mathematical operations

- Arithmetic operations (+, -, *, /, **, %):
  - x**y (x^y): 3**2 = 9
  - % (mod) : 3 % 2 = 1
- Comparisons :
  - 3==4 (False)
  - 3!=4 (True)
  - 3<4 (True)
  - 3>4 (False)
  - 3<=4 (True)
  - 3>=4 (True)
- Logical operations (and, or, not):
  - False and True (False)
  - False or True (True)
  - False or not True (False)

Warning 1

Not all operations work with all types

Warning 2

Output of operations depend on types

- 'Alan'+'Turing'='AlanTuring'
- 3 + True = 4
- True + True = 2
- 'Alan'+3=? (error)

Arrays

- Arrays are lists that can hold multiple values:
  - a=['John', 'Steven', 'Mark']
  - b=[3.5, True, 'Steven', -23]
- Elements of arrays can be accessed using their index:
  - indices start at '0'
  - a[0]='John',a[1]='Steven'
  - negative indices: a[-1]='Mark'
  - slicing: a[:2]=['John','Steven']

Arrays

- Arrays are lists that can hold multiple values:
  - a=[’John’, ’Steven’, ’Mark’]
  - b=[3.5, True, ’Steven’, -23]
- Elements of arrays can be accessed using their index:
  - indices start at ’0’
  - a[0]=’John’,a[1]=’Steven’
  - negative indices: a[-1]=’Mark’
  - slicing: a[:2]=[’John’,’Steven’]
- Adding and removing elements:
  - a.append[’Alice’] (a=[’John’, ’Steven’, ’Mark’,’Alice’])
  - a.remove[’Mark’] (a=[’John’, ’Steven’, ’Alice’])
  - a.pop(1) (removes a[1] from a and returns a[1])
- Useful methods:
  - len(a):length of a
  - max(a), min(a): largest/smallest value
  - a.sort() : a is now sorted (smaller values first)
  - a.index(’Steven’) = 1

Strings

- In Python strings are treated as lists of characters.
  - a='To be, or not to be'
  - a[0]='T', a[:5]='To be'...
- Some useful methods:
  - a.upper()–> 'TO BE, OR NOT TO BE'
  - a.lower()–> 'to be, or not to be'
  - a.split()–> ['To', 'be,' ,'or', 'not', 'to', 'be']
  - a.split(',')–>['To be', ' or not to be']
  - a.index('o')–> 1
  - a.index('be')–> 3
  - a.strip(): removes any leading, and trailing whitespaces.

Dictionaries

- Dictionaries are key value pairs:
  - D={’age’:30, ’height’:180, ’dob’: ’13 Jan 1992’}
- Dictionary values can be accessed via keys:
  - D[’age’]=30
  - D.keys()=[’age’, ’height’, ’dob’]
  - D.values()=[30, 180, ’13 Jan 1992’]
- Updating dictionaries:
  - D[’eyecolor’]=’brown’
  - D.update(D2): now D also contains all entries from D2.

Sets

- Sets are unordered collection of unique elements.
    - set1={1, 2, 3, 4, 5}
    - set2={4, 5, ’Alice’, ’Bob’}
    - 1 in set1 –> True

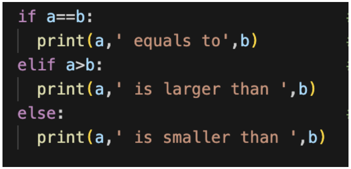
Sets

- Sets are unordered collection of unique elements.
  - set1={1, 2, 3, 4, 5}
  - set2={4, 5, 'Alice', 'Bob'}
  - 1 in set1 –> True
- Set operations:
  - set1.add() : adds an element to the set
  - set1.update(set2/list2): add set2 or list2 to set1
  - set1.union(set2): union of sets (set1+set2)
  - set1.intersection(set2): intersection of sets
  - set1.difference(set2): set1-set2

Control structures

Control structures impose conditions on the execution bits of code:

- if statements:
  - condition
  - code that is executed if condition is satified/True
- elif (else if) statements:
  - can be used to impose multiple conditions
  - checked only if previous if condition is not satisfied
- else:
  - what to do if *none* of the if/elif statements are satisfied.

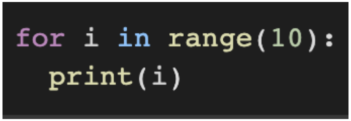


```
if a==b:
  print(a,' equals to',b)
elif a>b:
  print(a,' is larger than ',b)
else:
  print(a,' is smaller than ',b)
```

an if statement in Python

Loops

- Loops allow the repeated execution of bits of code.
  - indentation same as for if statements
- for loops are used to repeatedly execute commands over a range of values
  - range(n) : iterator from 0 to n-1
  - arrays and sets can also be used as iterators

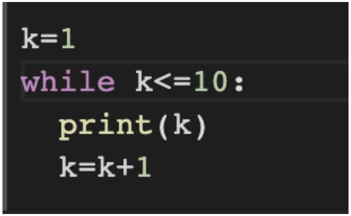


```
for i in range(10):
  print(i)
```

a for loop in Python

Loops

- while loops execute commands as long as a condition is satisfied
  - the condition should depend on the content of the loop
  - watch out for *infinite loops*!
- Loops can be nested and combined
- Control statements for loops:
  - break : stops the loop
  - continue: go back to the start of the loop



```
k=1
while k<=10:
  print(k)
  k=k+1
```

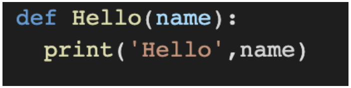
a while loop in Python

Lecture 3: Functions and classes

- We introduced **functions** in Python
- We learned how to define **classes** in Python
- and class **attributes** and **methods**

Functions

- A function is a reusable block of code that performs a specific task.
- Functions help with modularity and code reusability:
  - accept input variables/parameters
  - return values
  - modify input variables

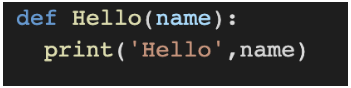


```
def Hello(name):
  print('Hello',name)
```

a function in Python

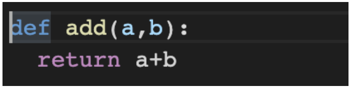
Functions

- A function is a reusable block of code that performs a specific task.
- Functions help with modularity and code reusability:
  - accept input variables/parameters
  - return values
  - modify input variables
- functions can be nested



```python
def Hello(name):
  print('Hello',name)
```

a function in Python



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

another function in Python

Defining and Calling Functions

- Define a function using the **def** keyword.
  - the body/content of the function is determined by indentation.
  - use **return** to specify the output of the function.
- Call the function by using its name and providing arguments.

Example

```
def greet(name):
        return "Hello, " + name
print(greet("Alice"))
# Output:  Hello, Alice
```

Scope of Variables

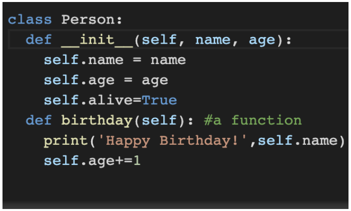
- Variables created inside a function are **local** to that function.
- Global variables are accessible throughout the entire script.
- Best practice: avoid modifying global variables inside functions.

Example

```
def add(x, y):
        result = x + y
        return result
print(add(2, 3))
# Output:  5
```

Object oriented programing

- Python is an object oriented programming language.
  - arrays, dictionaries, sets are Python objects
  - each type of object has their own properties and methods
- In Python we can define new classes/object types with their own:
  - properties
  - methods/functions

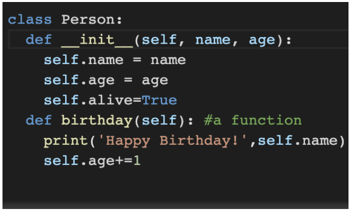


```python
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
    self.alive=True
  def birthday(self): #a function
    print('Happy Birthday!',self.name)
    self.age+=1
```

a class in Python

What is a Class?

- A class is a blueprint for creating objects (instances).
- Objects represent entities with specific attributes and behaviors.

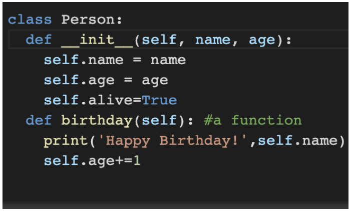


```python
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
    self.alive=True
  def birthday(self): #a function
    print('Happy Birthday!',self.name)
    self.age+=1
```

a class in Python

What is a Class?

- A class is a blueprint for creating objects (instances).
- Objects represent entities with specific attributes and behaviors.
- Syntax:
  - in Python new classes are defined using the **class** command.
  - **__init__()** : defines initialization routine.
  - **Attributes** are variables within a class, unique to each instance.
  - **Methods** are functions defined inside a class.



```python
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
    self.alive=True
  def birthday(self): #a function
    print('Happy Birthday!',self.name)
    self.age+=1
```

a class in Python

Lecture 4: Modules and file handling basics

- we introduced modules in Python
- we learned how to import modules
- In-built modules (math, random, os ...)
- Creating our own modules
- Installing 3rd party libraries
- Creating Python environments
- Opening files in ’r’, ’w’, ’a’ and ’b’
- Reading files and lines
- Writing and appending to files

What Are Modules?

Definition:

- Modules are reusable pieces of Python code.
- They can be built-in or user-defined.
- Allow for organized and efficient code reuse.

Examples:

- Built-in: `math`, `os`, `random`.
- User-defined: A Python file you create with functions and classes.

Using Built-in Modules

How to Import a Module:

- `import module_name`
- `from module_name import specific_item`
- `import module_name as alias`

Examples:

```
import math
print(math.sqrt(16))

from random import randint
print(randint(1, 10))

import os as operating_system
print(operating_system.getcwd())
```

Creating a User-Defined Module

Steps to Create a Module:

1. Create a Python file (e.g., `mymodule.py`).
2. Define functions, variables, or classes in it.
3. Import the file into your script.

Example: `mymodule.py`

```
def greet(name):
    return f"Hello, {name}!"
```

`main.py`

```
import mymodule

print(mymodule.greet("Alice"))
```

Installing Third-Party Libraries

What Are Third-Party Libraries?

- Libraries are collections of modules designed for specific tasks.
- Examples:
  - `numpy` for numerical computations.
  - `matplotlib` for data visualization.
  - `requests` for web requests.

Installing Libraries with `conda`:

- Command: `conda install library_name`.
- Example: `conda install numpy`.

Installing Libraries from PyPI:

- Use `pip` within an active `conda` environment.
- Command: `pip install library_name`.
- Example: `pip install requests`.

Viewing Installed Libraries:

```
conda list
```

File Handling Basics

Key Concepts:

- Files are opened with the `open()` function.
- Modes:
  - `’r’`: Read mode (default).
  - `’w’`: Write mode (overwrites existing content).
  - `’a’`: Append mode.
  - `’b’`: Binary mode.
- Use the `with` statement to handle files safely.

Reading Text Files

Basic File Reading Methods:

- `file.read()`: Reads the entire file.
- `file.readline()`: Reads one line at a time.
- `file.readlines()`: Reads all lines into a list.

Example:

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)

with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())
```

Writing to Text Files

Writing Methods:

- `file.write(string)`: Writes a string to the file.
- `file.writelines(list)`: Writes a list of strings to the file.

Example: Writing Data to a File

```
with open('output.txt', 'w') as file:
    file.write('Hello, World!\n')

data = ['Line 1\n', 'Line 2\n', 'Line 3\n']
with open('output.txt', 'a') as file:
    file.writelines(data)
```

Working with File Paths

Using the `os` Module:

- Check if a file exists: `os.path.exists(filepath)`.
- Get the current directory: `os.getcwd()`.
- Create directories: `os.makedirs()`.

Example:

```
import os

if os.path.exists('example.txt'):
    print('File exists!')

print(f'Current directory: {os.getcwd()}')

os.makedirs('new_folder', exist_ok=True)
```

Lecture 5: Pandas and Seaborn

Pandas: Data Analysis and Manipulation

- Core Data Structures: `Series`, `DataFrame`.
- Data Import and Export: Functions like `read_csv()`, `to_csv()`.
- Data Exploration: `head()`, `info()`, `describe()`.
- Data Preprocessing: Handling missing data, filtering rows, and modifying columns.
- Grouping and Aggregation: Using `groupby()` for insights.
- Statistical Functions: `mean()`, `median()`, `std()`.

Introduction to `pandas`

Key Functionalities:

- Data manipulation and analysis
- Handles structured data like CSVs, Excel files, and SQL queries
- Tools for cleaning, transforming, and summarizing data

Core Data Structures:

- `DataFrame`: 2D labeled table, similar to a spreadsheet
- `Series`: 1D labeled array, representing a single column or row

Learn more: pandas.pydata.org

Loading and Saving Data with `pandas`

Loading Data:

- `pd.read_csv('file.csv')`: Reads a CSV file into a DataFrame.
- `pd.read_excel('file.xlsx')`: Reads an Excel file.
- `pd.read_sql(query, connection)`: Reads data from a SQL database.
- `pd.read_json('file.json')`: Reads JSON data into a DataFrame.

Example:

```
# Load CSV file
data = pd.read_csv('file.csv')

# Save as Excel
data.to_excel('file.xlsx', index=False)
```

Loading and Saving Data with `pandas`

Saving Data:

- `data.to_csv('file.csv')`: Saves the DataFrame as a CSV file.
- `data.to_excel('file.xlsx')`: Exports the DataFrame to an Excel file.
- `data.to_json('file.json')`: Saves the DataFrame in JSON format.
- `data.to_sql('table_name', connection)`: Writes the DataFrame to a SQL table.

Example:

```
# Load CSV file
data = pd.read_csv('file.csv')

# Save as Excel
data.to_excel('file.xlsx', index=False)
```

Inspecting Data with `pandas`

Basic Inspection Methods:

- `data.head(n)`: Displays the first `n` rows of the DataFrame (default is 5).
- `data.tail(n)`: Displays the last `n` rows of the DataFrame.
- `data.info()`: Provides a summary of the DataFrame, including column types, counts, and memory usage.
- `data.describe()`: Calculates summary statistics for numeric columns.
- `data.shape`: Returns the dimensions of the DataFrame (`rows, columns`).

Inspecting Data with `pandas`

Example:

```
# Inspect the data
print(data.head())
print(data.info())
print(data.describe())

# Check the shape
print(f"Rows: {data.shape[0]}, Columns: {data.shape[1]}")
```

Usage:

- `head()` and `tail()`: Quick look at the beginning or end of the dataset.
- `info()`: Understand data types and missing values.
- `describe()`: Get key statistics like mean, median, and standard deviation.

Understanding Indices and Slicing

Code Example:

```
# Access a single column
column = data['column_name']

# Access rows by index
row = data.iloc[0]

# Slice rows
subset = data[10:20]
```

Concepts:

- **Indices**: Every row and column in a DataFrame has an index.
- **Slicing**: Use square brackets to slice rows or select columns.
- `data.iloc`: Access rows by their integer index position.
- `data['column_name']`: Access a column as a Series.

Basic Statistical Functions in `pandas`

Summary Statistics:

- Mean of a column:

```
mean_value = data['col'].mean()
```

- Standard deviation:

```
std_dev = data['col'].std()
```

- Median:

```
median_value = data['col'].median()
```

- Maximum and minimum values:

```
max_value = data['col'].max()
min_value = data['col'].min()
```

Basic Statistical Functions in `pandas`

Aggregations:

- Sum of values:
  ```
  total = data['col'].sum()
  ```
- Count of non-null values:
  ```
  count = data['col'].count()
  ```

Other Useful Functions:

- Correlation:
  ```
  correlation = data.corr()
  ```
- Unique values in a column:
  ```
  unique_values = data['col'].unique()
  ```
- Value counts (frequencies):
  ```
  value_counts = data['col'].value_counts()
  ```

Modifying Data with `pandas`

Adding and Updating Columns:

- Add a new column:
  ```
  data['new_col'] = data['col1'] + data['col2']
  ```
- Update values conditionally:
  ```
  data.loc[data['col1'] > 10, 'col1'] = 10
  ```

Dropping Rows and Columns:

- Drop a column:
  ```
  data = data.drop('col_name', axis=1)
  ```
- Drop rows by index:
  ```
  data = data.drop(index=[0, 1])
  ```

Renaming Columns:

- `data = data.rename(columns={'old_name': 'new_name'})`

Data Preprocessing with `pandas`

Handling Missing Data:

- Drop rows with missing values:

```
data = data.dropna()
```

- Fill missing values:

```
data['col'] = data['col'].fillna(value)
```

Data Transformation:

- Apply a function to a column:

```
data['col'] = data['col'].apply(myfunction)
```

- Normalize numeric columns:

```
 (data['col'] - data['col'].mean()) / data['col'].std()
```

Handling Duplicates:

- Remove duplicate rows:

```
data = data.drop_duplicates()
```

`groupby()` in Pandas

- `groupby()` is a powerful function in Pandas used for grouping data based on one or more columns.
- It is often used in combination with aggregation functions (e.g., `mean()`, `sum()`, `count()`).

How Does It Work?

- Splits the data into groups based on the values of a column(s).
- Applies a function (e.g., aggregation or transformation) to each group.
- Combines the results into a new DataFrame or Series.

Example: Calculate Mean Fare by Passenger Class (Titanic Dataset)

```
# Group by 'Pclass' and calculate mean fare
grouped_data = data.groupby('Pclass')['fare'].mean()
```

Output:

```
Pclass
1     84.154687
2     20.662183
3     13.675550
```

Lecture 5: Pandas and Seaborn

Seaborn: Data Visualization

- High-Level Interface: Simplified plotting for statistical graphics.
- Common Plot Types: Scatter plots, bar plots, histograms, box plots.
- Customization: Themes, color palettes, and labels.
- Integration with Pandas: Directly using DataFrames for visualization.

Lecture 6: Machine learning basics with sklearn

- **Supervised Learning**:
  - **Definition**: Models learn from labeled data to predict outputs for new inputs.
  - **Key Concepts**:
    - Classification (e.g., MNIST digit classification).
    - Algorithms: `KNeighborsClassifier`, `SVC`.
    - Train-Test Split: Preparing data for training and evaluation.

Lecture 6: Machine learning basics with sklearn

- **Unsupervised Learning**:
  - **Definition**: Models learn patterns and structures from unlabeled data.
  - **Key Concepts**:
    - Clustering (e.g., DBSCAN, K-Means on MNIST).
- **Scikit-learn Basics**:
  - Unified interface for training (`fit`) and predictions (`predict`).
  - Tools for preprocessing, model evaluation, and visualization.

Introducing Supervised Learning

What is Supervised Learning?

- A machine learning paradigm where the model learns from labeled data (features and their corresponding targets).
- Goal: Predict outcomes (labels) for unseen data based on training data.

Example: MNIST Digit Classification

- Dataset: Handwritten digits (0–9) represented as 28x28 pixel images (flattened into 784 features).
- Task: Predict the correct digit based on the pixel values.

Introducing Supervised Learning

What is Supervised Learning?

- A machine learning paradigm where the model learns from labeled data (features and their corresponding targets).
- Goal: Predict outcomes (labels) for unseen data based on training data.

Example: MNIST Digit Classification

- Dataset: Handwritten digits (0–9) represented as 28x28 pixel images (flattened into 784 features).
- Task: Predict the correct digit based on the pixel values.

K Nearest Neighbour (KNN)

- KNN is a simple, yet powerful algorithm for classification.
- It works by finding the 'k' closest data points to a new observation and assigning the most common label among them.
- KNN is often a good starting point for classification problems.

Example: KNN Classification on MNIST

Step 1: Import Libraries

- Import necessary libraries for loading data, preprocessing, and modeling.

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
```

Example: KNN Classification on MNIST

Step 2: Load the MNIST Dataset

- MNIST dataset contains 70,000 28x28 grayscale images of handwritten digits.

```
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"], mnist["target"]
```

Example: KNN Classification on MNIST

Step 3: Split the Data into Training and Testing Sets

- Split the data into 80% training and 20% testing.

```
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2)
```

Example: KNN Classification on MNIST

Step 4: Initialize the KNN Classifier and Train the Model

- Use KNN with 3 neighbors (k=3) to classify the digits.

```
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
```

Example: KNN Classification on MNIST

Step 5: Evaluate the Model

- Predict labels for the test set and calculate the accuracy.

```
y_pred = knn.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

- sk-learn contains a variety of models for supervised classification.
- all such models can used following the same general `.fit()` and `.predict()` routine.

Summary of Supervised Machine Learning

- **Supervised Learning Overview**:
  - The model is trained on input-output pairs, where the output is known (labeled data).
- **Key methods in sk-learn**:
  - `train_test_split(X, y)` is used to split the data into training and testing sets.
  - `model.fit(X_train, y_train)` is used to train the model on the training data.
  - `model.predict(X_test)` is used to make predictions on new, unseen data using the trained model.

Introduction to Unsupervised Learning

In unsupervised learning the model is trained using data that has no labeled outcomes or target variables. The goal is to identify underlying patterns, structures, or relationships within the data.

- **No Labeled Data**: Unlike supervised learning, where the model learns from labeled input-output pairs, unsupervised learning works with unlabeled data, aiming to find hidden structures or representations.
- **Applications**:
  - **Clustering**: Grouping similar data points together, for example, customer segmentation or document clustering.
  - **Dimensionality Reduction**: Reducing the number of features while retaining important information
  - **Anomaly Detection**: Identifying unusual or rare data points
- **Key Challenge**: evaluating model performance without explicit feedback. The interpretation of the results often requires domain knowledge.

K-Means Clustering

K-Means is one of the most widely used clustering algorithms. It aims to partition the dataset into K clusters, minimizing the variance within each cluster.

- The algorithm assigns each data point to the nearest centroid.
- The centroids are updated iteratively to minimize the sum of squared distances between data points and their corresponding centroids.
- Suitable for problems with well-defined ’spherical’ clusters.

Implementing K-Means in Scikit-learn with the MNIST Dataset

In this example, we aim to cluster the MNIST dataset into 10 groups, corresponding to the 10 possible digits (0-9).

```
from sklearn.cluster import KMeans

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X = mnist["data"].values  # Image features

# Apply K-Means clustering with 10 clusters (for digits 0-9)
kmeans = KMeans(n_clusters=10, random_state=42)
kmeans.fit(X)
```

Lecture 7: Natural language processing with nltk

- **Introduction to NLP and NLTK:**
  - Overview of NLP tasks and common techniques.
  - Introduction to NLTK for text processing.
- **Text Processing:**
  - Loading and cleaning text data.
  - Tokenization and basic text preprocessing.
- **Lemmatization:**
  - Using NLTK's lemmatizer to reduce words to their base forms.
- **Sentiment Analysis:**
  - Using CountVectorizer to convert text to features.
  - Training a classifier (e.g., Logistic Regression) for sentiment prediction.
- **NLP is a diverse field with many applications**

What is NLP?

- **Definition:** Natural Language Processing (NLP) is a field of computer science that focuses on the computational analysis, modeling, and manipulation of natural language data.
- **Applications:**
  - Sentiment Analysis
  - Translation (e.g., Google Translate)
  - Search Engines
  - Chatbots and Virtual Assistants
- **Relevance to research:**
  - Analyzing historical documents
  - Understanding social media trends
  - Text mining in literature and archives

Introduction to NLTK

- **NLTK (Natural Language Toolkit):**
  - A comprehensive library for working with human language data (text).
  - Provides tools for:
    - Tokenization (splitting text into words or sentences).
    - Stemming and Lemmatization.
    - Part-of-Speech (POS) Tagging.
    - Parsing and Chunking.
    - Working with corpora (e.g., Gutenberg, WordNet).
  - Widely used in education and research.
- **Documentation:** https://www.nltk.org/

Loading Text Data

- Text data can come from various sources:
  - Local files (e.g., `.txt`, `.csv`).
  - Online resources or APIs.
  - Built-in datasets from libraries like NLTK.
- Importance of loading data correctly:
  - Ensures compatibility with preprocessing tools.
  - Facilitates efficient analysis.

Python Example:

```
from nltk.corpus import gutenberg print(gutenberg.fileids())
Raw text as a single string: text = gutenberg.raw('austen-em
print(text[:500])  First 500 characters of the text
```

Cleaning Text Data

- Raw text often needs cleaning before analysis:
  - Convert text to lowercase for uniformity.
  - Remove punctuation and special characters.
  - Handle whitespace and remove stopwords if necessary.
- Why is cleaning important?
  - Reduces noise in the data.
  - Improves accuracy of NLP tasks.

Python Example:

```
cleaned_text = text.lower() # Convert to lowercase

# Remove punctuation
punctuations = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
cleaned_text = cleaned_text.translate(
str.maketrans('', '', punctuations))
print(cleaned_text[:500])  # First 500 cleaned characters
```

What is Tokenization and Why is it Important?

- **Definition:** Tokenization is the process of breaking text into smaller units, called **tokens**.
- Tokens can be:
  - Words (word-level tokenization).
  - Sentences (sentence-level tokenization).
  - Characters (character-level tokenization).
- Tokenization is a crucial preprocessing step in NLP tasks such as:
  - Text classification.
  - Machine translation.
  - Sentiment analysis.
- Why Tokenize?
  - Provides structure to raw text by splitting it into analyzable units.
  - Simplifies tasks like frequency analysis, parsing, and syntactic analysis.
  - Helps handle multilingual text with different delimiters.

Tokenization in NLTK

- NLTK provides built-in functions for tokenization:
  - `word_tokenize` for word-level tokenization.
  - `sent_tokenize` for sentence-level tokenization.
- Example: Tokenizing text into words.

Python Example:

```
from nltk.tokenize import word_tokenize

text = "Natural Language Processing is amazing!"
tokens = word_tokenize(text)
print(tokens)
# Output:
['Natural', 'Language', 'Processing', 'is', 'amazing', '!']
```

Sentence Tokenization Example

- Sentence tokenization splits a text into individual sentences.
- Useful for tasks like summarization or sentiment analysis.

Python Example:

```
from nltk.tokenize import sent_tokenize

text = "Tokenization is fun. NLTK makes it easy!"
sentences = sent_tokenize(text)
print(sentences)
# Output: ['Tokenization is fun.', 'NLTK makes it easy!']
```

What is Part-of-Speech (POS) Tagging?

- **Definition:** POS tagging is the process of assigning grammatical categories (e.g., noun, verb, adjective) to each word in a text.
- POS tags provide insights into the structure and meaning of a sentence.
- Examples of common POS tags:
  - `NN` – Noun (e.g., "dog", "apple").
  - `VB` – Verb (e.g., "run", "is").
  - `JJ` – Adjective (e.g., "beautiful", "large").
  - `RB` – Adverb (e.g., "quickly", "very").

Why is POS tagging useful?

- Improves text processing tasks such as lemmatization, parsing, and information extraction.
- Provides the context necessary to interpret words accurately.

POS Tagging in NLTK

- NLTK provides the `pos_tag()` function to perform POS tagging.
- The function assigns a POS tag to each token in a sentence.

Python Example:

```
from nltk import pos_tag, word_tokenize

text = "The quick brown fox jumps over the lazy dog."
tokens = word_tokenize(text)
pos_tags = pos_tag(tokens)

print(pos_tags)
# Output:
# [('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'),
#  ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'),
#  ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]
```

What is Lemmatization?

- **Definition:** Lemmatization is the process of reducing a word to its dictionary form, or **lemma**, by considering its context and part of speech.
- Unlike stemming, lemmatization:
  - Produces valid words.
  - Considers the meaning of the word.
- Example:
  - "running" → "run".
  - "geese" → "goose".
  - "better" → "good".

Why Use Lemmatization?

- Preserves the meaning of words by reducing them to their canonical forms.
- Avoids ambiguity that might arise from stemming, which may produce non-words.
- Useful in:
    - Sentiment analysis.
    - Information retrieval.
    - Text classification.
- Examples:
    - "studies" → "study".
    - "better" → "good".

Lemmatization in NLTK

- NLTK uses `WordNetLemmatizer` for lemmatization.
- Lemmatization considers the part of speech (POS) to find the correct lemma.
- Example: Using `WordNetLemmatizer` in Python.

Python Example:

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

words = ["running", "geese", "better", "studies"]
lemmas = [lemmatizer.lemmatize(word) for word in words]

print(lemmas)
# Output: ['running', 'goose', 'better', 'study']
```

Sentiment Analysis

- Sentiment analysis is the process of identifying and classifying the emotional tone of a text.
- Common categories include:
  - **Positive**, **Negative**, **Neutral**
- Applications:
  - Analyzing customer reviews.
  - Monitoring social media sentiment.
  - Understanding public opinion.
- Example:
  - Text: *"This product is amazing!"*
  - Sentiment: **Positive**

Sentiment Analysis Pipeline

- Typical steps in a sentiment analysis pipeline:
  1. **Data Collection:** Gather text data (e.g., tweets, reviews).
  2. **Preprocessing:** Clean and tokenize the text.
  3. **Feature Extraction:** Convert text into numerical features (e.g., word counts, embeddings).
  4. **Classification:** Use a machine learning model to classify sentiment.
- Tools: NLTK, scikit-learn, pre-trained models (e.g., VADER, BERT).

Sentiment Analysis with CountVectorizer

- Converts text into a bag-of-words representation.
- Each text is represented as a vector where:
  - Columns correspond to unique words in the dataset.
  - Values are the frequency of each word in the text.
- Example:
  - Texts: `"I love cats"`, `"I love dogs"`

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

  - Columns: [`I`, `love`, `cats`, `dogs`]
- Converting text into numerical features using enables the application of standard ML algorithms.

Python Example:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression

# Sample dataset
texts = ["I love this product!", "This is terrible.",
"It’s okay."]
labels = [1, 0, 1]  # 1: Positive, 0: Negative

# Transform texts to bag-of-words
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(texts)

# Train classifier
classifier = LogisticRegression()
classifier.fit(X, labels)
```

Lecture 8: Version control with git

- **Version control essentials:**
  - Track changes to your files.
  - Revert to previous versions.
  - Experiment without fear.
- **Git basics:**
  - Staging and committing changes.
  - Branching for parallel development.
  - Viewing history and differences.
- **Collaboration with Git:**
  - Remote repositories (GitHub).
  - Cloning, pushing, and pulling.
  - Branching and merging.
  - Pull requests for code review.

What is Version Control?

- Like "track changes" in Word, but much more powerful!
- **Benefits:**
  - Track the history of your project.
  - Revert to previous versions.
  - Experiment without fear of losing work.
  - Collaborate seamlessly.

Installation and configuration

1. Download and install Git from the official website: `https://git-scm.com/`
2. Configure Git:
   - Set your name: ‘git config –global user.name "Your Name"‘
   - Set your email: ‘git config –global user.email "your.email@example.com"‘

Creating a git Repository

- **Repository:** A project folder tracked by Git.
  - Git can track multiple files of different types simultaneously.
  - Git can also track sub folders.
- **Initializing a git repo:**
  - Open your terminal/command prompt.
  - Navigate to your project folder.
  - Type ‘git init‘.

How Git Tracks Changes

Git uses a snapshot-based approach:

- Each commit is a snapshot of the entire project at a given point in time.
- Files that haven't changed are stored as links to the previous snapshot.

Core Components

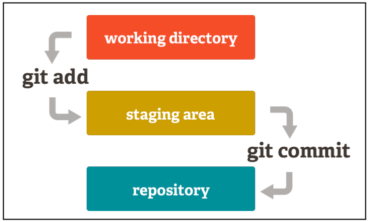
- **Objects:** Git stores everything as objects:
  - **Blob:** Stores file content.
  - **Tree:** Represents a directory, linking blobs and sub-trees.
  - **Commit:** Points to a tree and includes metadata (e.g., author, message).
- **Hashes:** Each object has a unique identifier (a SHA-1 hash).

Example:

```
Commit -> Tree -> Blob (file1)
                 -> Blob (file2)
                 -> Tree (subdir) -> Blob (file3)
```

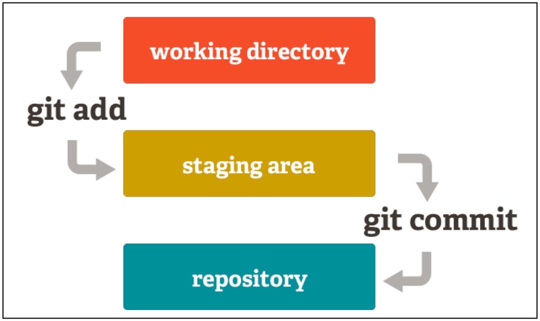
Staging and Committing Changes in Git

- **Staging Area:** A temporary holding area for changes.
  - Allows you to select which modifications should be included in the next commit.
  - Provides granular control over version history.
- **Committing:**
  - Creates a snapshot of the staged changes, permanently recording them in the repository.
  - Each commit is accompanied by a message explaining the changes made.



Basic Git Commands

Command	Description
‘git status‘	Check the status of your files
‘git add filename ‘	Stage changes for commit
‘git reset filename ‘	Unstage changes
‘git commit -m "message"‘	Record changes with a message
‘git log‘	View the commit history



Example: Tracking a Text File

1. **Create a file:** Create a new file named `paper.txt` and write the first paragraph of your paper in `paper.txt`.
2. **Initialize a repository:** Open your terminal, navigate to the folder containing `paper.txt`, and type `git init`.
3. **Check status:** Type `git status`. You'll see `paper.txt` listed as untracked.
4. **Stage changes:** Type `git add paper.txt`.
5. **Commit changes:** Type `git commit -m "First draft of introduction"`.
6. **Make more changes:** Add a second paragraph to `paper.txt`.
7. **Stage and commit again:** Repeat steps 4-6 with a new commit message, e.g., "Added second paragraph".
8. **View history:** Type `git log` to see your commit history.

Understanding ‘git log‘

- ‘git log‘ displays the commit history of your repository. Each commit entry includes:
  - **Commit hash:** A unique identifier for the commit (e.g., `a1b2c3d`).
  - **Author & Date:** The name and email of the person who made the commit and the date and time of the commit.
  - **Commit message:** The message provided when the commit was created.

Understanding ‘git log‘

- Example output:

```
commit a1b2c3d (HEAD -> main)
Author: Your Name <your.email@example.com>
Date:   Thu Jan 9 14:00:00 2025 +0100

    Added second paragraph

commit 7f6e5d4c
Author: Your Name <your.email@example.com>
Date:   Thu Jan 9 13:30:00 2025 +0100

    First draft of introduction
```

Comparing Changes with ‘git diff‘

- **See what’s changed:** ‘git diff‘ shows you the differences between various states of your project.
- **Common use cases:**
  - **‘git diff‘**: See the changes you’ve made but haven’t staged yet.
  - **‘git diff –staged‘**: See the changes you’ve staged but haven’t committed yet.
  - **‘git diff commit_hash commit_hash ‘**: See the differences between two commits.
- **Understanding the output:** ‘git diff‘ uses a standard format to highlight additions and deletions.
- **Benefits:**
  - Review your changes before committing.
  - Understand the impact of different commits.
  - Identify the source of bugs or errors.

Peeking into the Past with ‘git checkout‘

- **Time travel without consequences:** ‘git checkout‘ allows you to temporarily switch to a different point in your project’s history without altering the current state.
- **Identify the commit:** Use ‘git log‘ to find the commit hash you want to explore.
- **Checkout a commit:** ‘git checkout commit_hash ‘ will temporarily switch your project to the state it was in at that commit. You can examine the files as they were at that specific point in time.
- **Return to the present:** ‘git checkout branch name ‘ (usually ‘main‘ or ‘master‘) will bring you back to your current working state.
- **Important notes:**
  - ‘git checkout‘ is like a "read-only" view of the past. Any changes you make in this state won’t be saved unless you create a new branch.
  - It’s a powerful tool for inspecting previous versions, understanding how your project evolved, and even recovering lost code.

Reverting to a Previous Version

- **Mistakes happen! Git allows you to go back in time.**
- **Identify the commit:** Use ‘git log‘ to find the commit hash you want to revert to.
- **Revert with ‘git revert commit_hash ‘:**
  - This command **undoes the changes** introduced in the specified commit.
  - It analyzes the changes made in that commit and applies the **opposite changes** to your current files.
  - For example, if the commit added a paragraph, ‘git revert‘ will delete that paragraph. If the commit deleted a sentence, ‘git revert‘ will add it back.
  - Git then creates a **new commit** with these "undo" changes, keeping a clear record of your actions.
- **Important:**
  - Reverting doesn’t erase the original commit from history.
  - It adds a new commit that counteracts the changes, maintaining a complete and traceable history.

Resetting to a Previous State with ‘git reset‘

- **A more forceful way to go back in time:** ‘git reset‘ moves the HEAD pointer (which indicates your current position in the project history) to a specific commit.
- **How it works:**
  - It’s like rewinding your project’s history to a specific point.
  - By default, it doesn’t delete commits, but it makes them inaccessible through the normal ‘git log‘.
  - Think of it like removing pages from a table of contents; the pages still exist in the book, but you can’t easily find them anymore.
- **Use with caution!**: ‘git reset‘ can alter your project history, so it’s important to understand its implications.

Resetting to a Previous State with ‘git reset‘

- **Reset options:**
  - **‘git reset –soft commit_hash ‘**: Keeps your changes staged.
  - **‘git reset –mixed commit_hash ‘**: Unstages your changes (default).
  - **‘git reset –hard commit_hash ‘**: Discards all uncommitted changes and any commits after the specified commit.
- **Example:** ‘git reset –hard a1b2c3d‘ will reset your project to the state it was in at commit ‘a1b2c3d‘, discarding any changes made after that commit.
- **Warning:** Be extremely careful with ‘git reset –hard‘ as it permanently deletes any uncommitted changes and makes it difficult to recover the "lost" commits!

Example: Using ‘git revert‘ and ‘git reset‘

Scenario:

We have a file named ‘myfile.txt‘ with three lines added in separate commits.

1. **Revert the last commit:**

   ```
   git revert HEAD
   ```

   This undoes the changes introduced in the last commit by creating a new commit.

2. **Reset to the first commit:**

   ```
   git reset --hard HEAD~2
   ```

   This resets the repository to the state of the first commit, discarding the second and third commits and any unstaged changes. **Use with caution!**

3. **Verify the changes:**

   ```
   cat myfile.txt
   ```

   This will show that the file now only contains the first line.

What are Branches?

- **Independent lines of development:** Branches allow you to create separate versions of your project.
- **Like parallel universes:** Each branch can evolve independently without affecting other branches.
- **Experiment freely:** Branches provide a safe space to try new ideas, explore different approaches, or fix bugs without disrupting the main project.

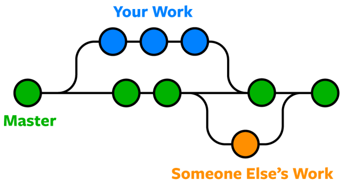


Figure: Branching in Git

Working with Branches

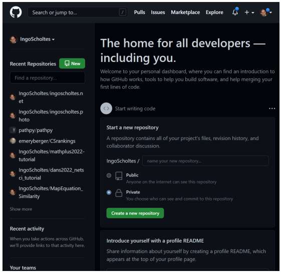
- **Create a new branch:** ‘git branch branch_name ‘ (e.g., ‘git branch feature-x‘)
- **Switch to a branch:** ‘git checkout branch_name ‘
- **List all branches:** ‘git branch‘ (the current branch will be highlighted)
- **Merge branches:** ‘git merge branch_name ‘ (merges the specified branch into the current branch)
- **Delete a branch:** ‘git branch -d branch_name ‘ (only after merging it)

Why Use Branches?

- **Feature development:** Isolate new features or experimental changes from the main codebase.
- **Bug fixing:** Create a dedicated branch to fix bugs without interrupting other work.
- **Collaboration:** Allow multiple people to work on different parts of the project simultaneously.
- **Versioning:** Maintain different versions of your project (e.g., for different publications or audiences).

What are Remote Repositories?

- **Repositories on a server:** Remote repositories are versions of your project stored on a server, like GitHub, GitLab, or Bitbucket.
- **Centralized hub:** They act as a central hub for collaboration, allowing multiple users to access and contribute to the same project.
- **Backup and sharing:** Remote repositories provide a backup of your work and enable easy sharing with others.



The home for all developers — including you.

Welcome to your personal dashboard, where you can find an introduction to how GitHub works, tools to help you build software, and help merging your first lines of code.

Start a new repository

A repository contains all of your project's files, revision history, and collaborator discussion.

IngoScholtes / name your new repository...

Public
Anyone on the internet can see this repository

[✓ Private]
You choose who can see and commit to this repository

Recent activity

When you take actions across GitHub, we'll provide links to that activity here.

Introduce yourself with a profile README

Share information about yourself by creating a profile README, which appears at the top of your profile page.

Figure: A remote repository

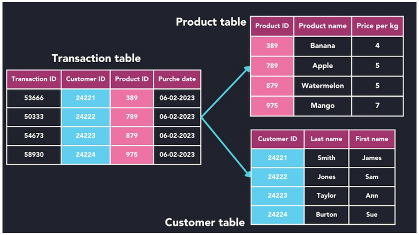
- **Clone a repository:**
  - ‘git clone repository_url ‘
  - Creates a local copy of the remote repository on your computer.
  - Example: ‘git clone https://github.com/username/project.git‘
- **Push your changes:**
  - ‘git push origin branch_name ‘
  - Uploads your local commits to the remote repository.
  - ‘origin‘ usually refers to the default remote (the original repository you cloned from).
  - Example: ‘git push origin my-feature-branch‘
- **Pull changes from others:**
  - ‘git pull origin branch_name ‘
  - Downloads and merges changes from the remote repository into your local branch.
  - Example: ‘git pull origin main‘

Lecture 9: Database basics with sqlite

- **Databases:** Organized collections of data for efficient storage, retrieval, and analysis.
- **Relational databases:** Data is organized in tables with rows and columns.
- **SQL:** A powerful language for interacting with relational databases.
- **SQLite:** A lightweight, file-based database system ideal for learning and small projects.
- **Key concepts:** Tables, columns, rows, primary keys, foreign keys, SQL commands (‘SELECT‘, ‘INSERT‘, ‘UPDATE‘, ‘DELETE‘).
- **Python and SQLite:** Python’s ‘sqlite3‘ module allows seamless interaction with SQLite databases.
- **Python integration:** Connecting to a database, creating tables, inserting data, querying data, and committing changes.

What is a Database?

- **Organized collection of data**
  - Digital filing cabinet
  - Library catalogue
  - Phone book



Product table

Product ID	Product name	Price per kg
389	Banana	4
789	Apple	5
879	Watermelon	5
975	Mango	7

Transaction table

Transaction ID	Customer ID	Product ID	Purche date
53666	24221	389	06-02-2023
50333	24222	789	06-02-2023
54673	24223	879	06-02-2023
58930	24224	975	06-02-2023

Customer table

Customer ID	Last name	First name
24221	Smith	James
24222	Jones	Sam
24223	Taylor	Ann
24224	Burton	Sue

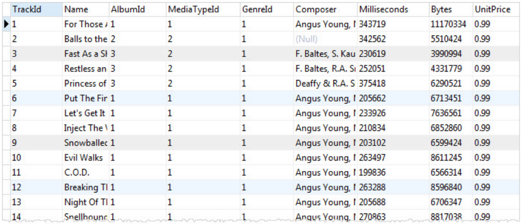
a database

Why Use Databases?

- **Efficient data storage and retrieval:**
  - Easily store and find the information you need.
- **Data integrity and consistency:**
  - Ensure data accuracy and prevent errors.
- **Reduced data redundancy:**
  - Avoid storing the same information multiple times.
- **Powerful querying and analysis:**
  - Extract meaningful insights from your data.

Relational Databases: Key Concepts

- **Tables:**
  - Organized structures that hold data in rows and columns.
  - Like spreadsheets or tables in a word processor.
- **Columns:**
  - Define the type of data stored (e.g., text, number, date).
  - Like the headers in a spreadsheet.
- **Rows:**
  - Represent individual records or entries in the table.



TrackId	Name	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice
1	For Those /	1	1	1	Angus Young, I	343719	11170334	0.99
2	Balls to the	2	2	1	(Null)	342562	5510424	0.99
3	Fast As a Sl	3	2	1	F. Baltes, S. Kau	230619	3990994	0.99
4	Restless an	3	2	1	F. Baltes, R.A. Sı	252051	4331779	0.99
5	Princess of	3	2	1	Deaffy & R.A. S	375418	6290521	0.99
6	Put The Fir	1	1	1	Angus Young, I	205662	6713451	0.99
7	Let's Get It	1	1	1	Angus Young, I	233926	7636561	0.99
8	Inject The \	1	1	1	Angus Young, I	210834	6852860	0.99
9	Snowballec	1	1	1	Angus Young, I	203102	6599424	0.99
10	Evil Walks	1	1	1	Angus Young, I	263497	8611245	0.99
11	C.O.D.	1	1	1	Angus Young, I	199836	6566314	0.99
12	Breaking Tl	1	1	1	Angus Young, I	263288	8596840	0.99
13	Night Of Tl	1	1	1	Angus Young, I	205688	6706347	0.99
14	Spellbounc	1	1	1	Angus Young, I	270863	8817038	0.99

Relational Databases: More Key Concepts

- **Primary Key:**
  - Uniquely identifies each row in a table.
  - Ensures that each record can be distinguished.
  - Often an auto-incrementing integer.
- **Foreign Key:**
  - A column that links related data in different tables.
  - Creates relationships between tables.
  - Refers to the primary key of another table.

id (PK)	name	nationality	birth_year
1	Jane Austen	English	1775
2	Charles Dickens	English	1812

Table 1: Authors

id (PK)	title	author_id **(FK)**
1	Pride and Prejudice	1
2	Oliver Twist	2

Table 2: Books

SQL Basics: Querying Data with SELECT

- **SQL (Structured Query Language):**
  - A language for interacting with relational databases.
- **SELECT statement:**
  - Used to retrieve data from a table.
  - Basic structure:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- **Explanation:**
  - `SELECT`: Specifies the columns you want to retrieve. Use '*' to select all columns.
  - `FROM`: Specifies the table you want to retrieve data from.
  - `WHERE`: (Optional) Specifies a condition to filter the results.
- **Example:**

```
SELECT title, author FROM books WHERE year > 1900;
```

  - This query retrieves the title and author of books published after 1900 from the "books" table.

SQL Basics: Modifying Data

- **INSERT statement:**
  - Used to add new data to a table.
- **Example:**

```
INSERT INTO books (title, author, year)
VALUES ('1984', 'George Orwell', 1949);
```

- **UPDATE statement:**
  - Used to modify existing data in a table.
- **Example:**

```
UPDATE books
SET year = 1948
WHERE title = '1984';
```

- **DELETE statement:**
  - Used to remove data from a table.
- **Example:**

```
DELETE FROM books WHERE id = 10;
```

The `sqlite3` Module

- **Built-in module:** Python comes with a built-in module called `sqlite3` for working with SQLite databases.
- **No installation needed:** You don't need to install any external libraries. Just **import sqlite3**.
- **Simple to use:** The `sqlite3` module provides a straightforward way to interact with SQLite databases using Python code.

Connecting to a Database

- **Establish a connection:** Use the `connect()` function to create a connection to the database.
- **Database file:** If the database file doesn't exist, it will be created.
- **Example:**

```
import sqlite3

conn = sqlite3.connect('mydatabase.db')
```

  - This creates a connection to the database file "mydatabase.db".
  - To execute SQL queries and fetch results we need a **cursor** object:

```
cursor = conn.cursor()
```

Using the Cursor

- **The Cursor Object:** The `cursor` object is essential for executing SQL queries and fetching results. Think of it as a pointer that allows you to traverse and manipulate data within the database. It is created using `conn.cursor()`.
- It is crucial to close the connection using `conn.close()` when you are finished interacting with the database to release resources.
- The cursor object has methods to execute SQL queries and retrieve data.
- Some important methods include:
  - `cursor.execute(sql_query)`: Executes an SQL query.
  - `cursor.fetchone()`: Fetches the next row of a query result.
  - `cursor.fetchall()`: Fetches all remaining rows of a query result.
  - `conn.commit()`: Saves changes to the database.

Creating a Table

- **Execute SQL:** Use the `execute()` method of the cursor to execute SQL statements.
- **Commit changes:** Use `conn.commit()` to save the changes to the database.
- **Example:**

```
import sqlite3

conn = sqlite3.connect('mydatabase.db')
cursor = conn.cursor()

cursor.execute('''CREATE TABLE IF NOT EXISTS books
            (id INTEGER PRIMARY KEY, title TEXT,
            author TEXT, year INTEGER)''')
conn.commit()
```

Committing Changes to the Database

- **Database transactions:** Changes made with `INSERT`, `UPDATE`, or `DELETE` are not immediately saved to the database file.
- **Committing changes:** Use `conn.commit()` to save the changes permanently.

Example:

```
cursor.execute("INSERT INTO books (title, author, year)
VALUES (?, ?, ?)",
('The Restaurant at the End of the Universe',
'Douglas Adams', 1980))
conn.commit()
```

- Without `conn.commit()`, the new book would not be saved in the database.

Fetching Data

- **Execute a SELECT query:** Use `cursor.execute()` with a SELECT statement.
- **Fetch methods:**
  - `fetchone()`: Retrieves the next row of the result as a tuple.
  - `fetchall()`: Retrieves all rows of the result as a list of tuples.
  - `fetchmany(size)`: Retrieves the specified number of rows.

Example:

```
cursor.execute("SELECT * FROM books WHERE author=?",
('Douglas Adams',))

# Fetch one row
first_row = cursor.fetchone()
print(first_row)

# Fetch all rows
all_rows = cursor.fetchall()
```

Example: Authors and Books Database

- **Two tables:**
  - `authors`: Stores information about authors (id, name, nationality, birth_year).
  - `books`: Stores information about books (id, title, author_id).
- **Foreign key:** The `author_id` column in the `books` table is a foreign key that references the `id` column in the `authors` table.
- **Creating the tables:**

```
cursor.execute('''CREATE TABLE IF NOT EXISTS authors
        (id INTEGER PRIMARY KEY, name TEXT,
        nationality TEXT, birth_year INTEGER)''')

cursor.execute('''CREATE TABLE IF NOT EXISTS books
        (id INTEGER PRIMARY KEY, title TEXT,
        author_id INTEGER,
        FOREIGN KEY (author_id) REFERENCES authors (id))''
conn.commit()
```

Example: Inserting Data

- **Insert authors:**

```
cursor.execute("INSERT INTO authors
(name, nationality, birth_year) VALUES (?, ?, ?)",
('Jane Austen', 'English', 1775))
cursor.execute("INSERT INTO authors
(name, nationality, birth_year) VALUES (?, ?, ?)",
('Charles Dickens', 'English', 1812))
conn.commit()
```

- **Insert books:**

```
cursor.execute("INSERT INTO books (title, author_id)
VALUES (?, ?)", ('Pride and Prejudice', 1))
cursor.execute("INSERT INTO books (title, author_id)
VALUES (?, ?)", ('Oliver Twist', 2))
conn.commit()
```

Example: Querying with Joins

- **JOIN clause:** Used to combine data from multiple tables based on related columns.
- **Retrieve book titles and author names:**

```
cursor.execute('''SELECT books.title, authors.name
    FROM books
    INNER JOIN authors ON books.author_id = authors.id''')
results = cursor.fetchall()
for row in results:
    print(row)
```

- **Output:**

```
    ('Pride and Prejudice', 'Jane Austen')
    ('Oliver Twist', 'Charles Dickens')
```

Lecture 10: Parallel processing

- **Parallel processing** uses multiple CPU cores to execute parts of a program simultaneously, leading to significant performance improvements for CPU-bound tasks.
- Processes are independent execution environments with their own memory spaces.
- The `multiprocessing` module in Python provides tools for creating and managing processes.
- `multiprocessing.Process` creates new processes.
- `process.start()` starts a process, and `process.join()` waits for it to finish.
- `multiprocessing.Queue` is a simple way to share data between processes (Inter-Process Communication).
- `multiprocessing.Pool` provides a higher-level interface for managing a pool of worker processes and simplifies common parallel tasks.

- **What is Parallel Processing?**
  - Executing multiple parts of a program **simultaneously** on multiple CPU cores.
  - In contrast sequential processing executes instructions one after the other.
- **Why is it Important?**
  - Significantly speed up computationally intensive tasks.
  - Efficient use of modern multi-core processors.
- **Real-World Examples:**
  - Scientific simulations (e.g., weather forecasting, physics simulations).
  - Large-scale data analysis and processing.
  - Machine Learning model training especially Artificial Neural Networks.
  - Image and video processing.

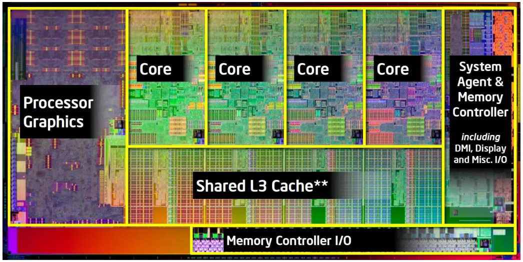
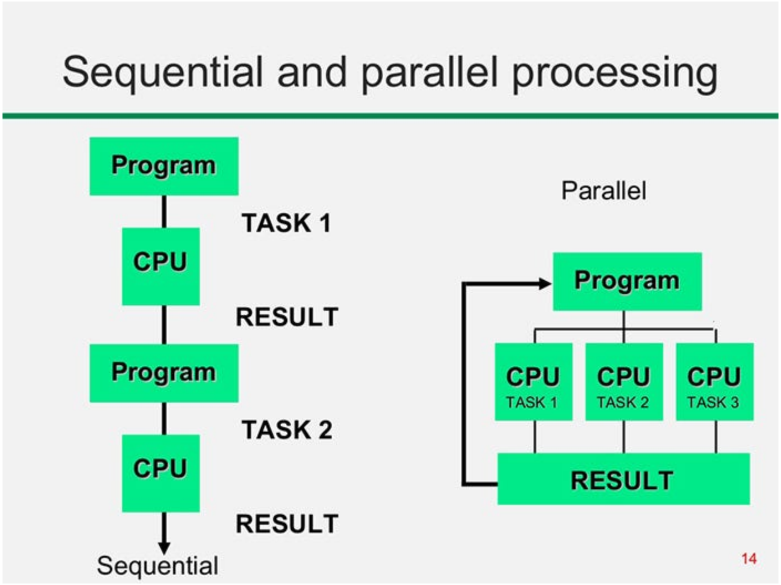


Figure: A multi core CPU



Sequential and parallel processing

Processes and the Operating System

- **What is a Process?**
  - An **independent** execution environment.
  - Has its own memory space, resources (files, open sockets, etc.), and process ID (PID).
  - Managed by the Operating System (OS).
- **Operating System and Process Management:**
  - The OS is responsible for:
    - Creating and terminating processes.
    - Allocating resources to processes.
    - Scheduling processes to run on CPU cores (process scheduling).
  - Processes are isolated from each other. Changes in one process do not affect other processes (by default).

Process Creation

- When you start a program, the OS creates a new process for it.
- The program's code, data, and other resources are loaded into the process's memory space.
- The OS then schedules the process to run on a CPU core.
- In a multi-core system, multiple processes can run **simultaneously**.

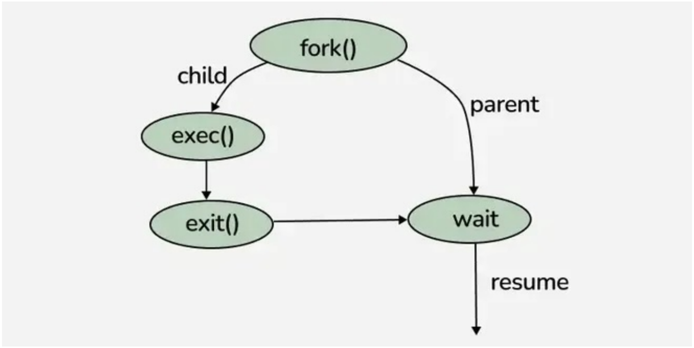


Figure: Simplified Process Creation

Introduction to `multiprocessing` in Python

- The `multiprocessing` module provides tools for creating and managing processes in Python.
- Key components:
  - `multiprocessing.Process`: Represents a process.
  - `process.start()`: Starts the process.
  - `process.join()`: Waits for the process to finish.

```
import multiprocessing
import time
def task(name):
    print(f"Process {name}: Starting")
    time.sleep(1) # Simulate some work
    print(f"Process {name}: Finishing")
if __name__ == "__main__":
    p = multiprocessing.Process(target=task,
args=("My Process",))
    p.start()
    p.join()
    print("Main process finished")
```

- `target`: The function that the process will execute.
- `args`: Arguments passed to the target function (must be a tuple, even for a single argument).
- The `if __name__ == "__main__":` block is crucial to prevent recursive process creation on Windows.

Creating Multiple Processes

```
def task(name):
    print("Process ",name," Starting")
    time.sleep(1) # doing something
    print("Process ",name," Finishing")
if __name__ == "__main__":
    processes = []
    for i in range(3):
        p = multiprocessing.Process(target=task, args=[i])
        processes.append(p)
        p.start()
    for p in processes:
        p.join()
    print("Main process finished")
```

- The `processes` list keeps track of the created processes so we can wait for them to finish using `join()`.
- Each process executes the `task` function independently.

Data Sharing Between Processes

- Processes have separate memory spaces. This means they cannot directly access each other's variables.
- To share data between processes, we need Inter-Process Communication (IPC) mechanisms.
- `multiprocessing.Queue` provides a simple way to exchange data between processes.
- **Using `multiprocessing.Queue`:**
  - Create a `Queue` object: `q = multiprocessing.Queue()`
  - Put data into the queue from a process: `q.put(data)`
  - Get data from the queue in another process (or the main process): `data = q.get()`

Process Communication with Queue

```
import multiprocessing

def worker(q):
    q.put("Hello from process!")

if __name__ == "__main__":
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=worker, args=(q,))
    p.start()
    message = q.get()
    p.join()
    print("Message from process: ",message)
```

- The `worker` function puts a message into the queue.
- The main process retrieves the message from the queue using `q.get()`.
- `q.get()` is a **blocking** operation. The main process will wait until data is available in the queue.

```
import multiprocessing
def square(num, q):
    q.put(num * num)
if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
    q = multiprocessing.Queue()
    processes = []
    for num in numbers:
        p = multiprocessing.Process(target=square,
args=(num, q))
        processes.append(p)
        p.start()
    results = []
    for p in processes:
        results.append(q.get()) #retreive results
        p.join()
    print(results)
```

`multiprocessing.Pool`

- Managing processes individually (starting, joining, handling queues) can become cumbersome, especially for complex parallel tasks.
- `multiprocessing.Pool` provides a simpler interface for managing a pool of worker processes.
- It automatically distributes tasks across the available processes and collects the results.
- Key methods:
  - `pool.map(function, iterable)`: Applies a function to each item in an iterable, distributing the work across the process pool. Returns a list of results in the same order as the input iterable.
  - `pool.apply_async(function, args)`: Applies a function asynchronously. Returns an `AsyncResult` object, which can be used to retrieve the result later using `get()`.

Parallel Calculation with `Pool`

```
import multiprocessing
def square(num):
    return num * num
if __name__ == "__main__":
    pool=multiprocessing.Pool(processes=4)
    numbers = [1, 2, 3, 4, 5]
    results = pool.map(square, numbers)
    print(results)
    pool.close()
```

- The `pool=multiprocessing.Pool(processes=4)` statement creates a process pool.
- `pool.map()` returns results is a list with the results in the same order as the input.
- `pool.close()` closes the pool.

Parallel Calculation with `Pool`

```
import multiprocessing
def square(num):
    return num * num
if __name__ == "__main__":
    pool=multiprocessing.Pool(processes=4)
    numbers = [1, 2, 3, 4, 5]
    async_results = [pool.apply_async(square,
    (num,)) for num in numbers]
    async_results_list = [res.get() for res
    in async_results]
    print(async_results_list)
    pool.close()
```

- `pool=multiprocessing.Pool(processes=4)` creates a process pool.
- `pool.apply_async()` shows how to launch processes asynchronically and retrieve the results later.
- `pool.close()` closes the pool.

Final Recap

- **In 10 Lectures we only covered the basics.**

Final Recap

- **In 10 Lectures we only covered the basics.**
- **But plenty of possibilities for further development and applications!**

Final Recap

- **In 10 Lectures we only covered the basics.**
- **But plenty of possibilities for further development and applications!**
- **Foundations for exploration and applying programming in your studies, research and beyond.**

Final Recap

- **In 10 Lectures we only covered the basics.**
- **But plenty of possibilities for further development and applications!**
- **Foundations for exploration and applying programming in your studies, research and beyond.**

Good luck in the exam and beyond!