Python Basics for AI: Self-Study Guide

Introduction

This self-study guide covers essential Python concepts required for the AI course. Python is the primary programming language used in AI and machine learning due to its simplicity, readability, and extensive libraries. This document will help you get up to speed with Python fundamentals before diving into AI concepts.

1. Python Installation and Setup

Installing Python

1. Visit python.org and download the latest version (Python 3.9+ recommended)
2. Follow the installation instructions for your operating system
3. Verify installation by opening a terminal/command prompt and typing:

   python --version

Setting Up a Development Environment Using VS Code

1. Download and install Visual Studio Code
2. Install the Python extension from the VS Code marketplace

Essential Libraries for AI

Install these core libraries:

pip install requests

2. Basic Syntax and Data Types

Python Syntax Basics

- Python uses **indentation (whitespace)** to define code blocks
- Statements don't need semicolons
- Comments use the **# symbol**

```
# This is a comment
print("Hello World")  # This is also a comment

if True:
    print("Indentation matters in Python")
    print("This is still part of the if block")
print("This is outside the if block")
```

No Copy Paste – Have to type

Variables and Assignment

- Variables **don't** need **explicit** declaration
- Dynamic typing (type is determined at runtime)

```
x = 10          # Integer
name = "Alice"  # String
pi = 3.14       # Float
is_valid = True # Boolean

# Multiple assignment
a, b, c = 1, 2, 3

# Type checking
print(type(x))      # <class 'int'>
print(type(name))   # <class 'str'>
```

Basic Data Types

Numeric Types:

```
# Integer
x = 42

# Float
y = 3.14159

# Complex number
z = 1 + 2j

# Basic operations
print(x + y)  # Addition: 45.14159
print(x * y)  # Multiplication: 131.94678
print(x / 5)  # Division: 8.4 (returns float)
print(x // 5) # Integer division: 8
print(x % 5)  # Modulus: 2
print(x ** 2) # Exponentiation: 1764
```

Strings:

```
# String creation
s1 = 'Single quotes'
s2 = "Double quotes"
s3 = """Triple quotes for
multi-line strings"""

# String operations
name = "Python"
print(len(name))        # Length: 6
print(name[0])          # Indexing: 'P'
print(name[1:4])        # Slicing: 'yth'
print(name + " language") # Concatenation: 'Python language'
print(name * 3)         # Repetition: 'PythonPythonPython'
print("th" in name)     # Membership: True

# String methods
print(name.upper())     # 'PYTHON'
print(name.replace('n', 'x')) # 'Pythox'
print("  hello  ".strip())    # 'hello'
```

Boolean Type:

```
x = True
y = False

# Boolean operations
print(x and y)  # False
print(x or y)   # True
print(not x)    # False

# Comparison operators
print(5 > 3)    # True
print(5 == 5)   # True
print(5 != 5)   # False
```

None Type:

```
x = None
print(x is None)  # True
```

3. Control Structures

Conditional Statements

```python
x = 10

# If statement
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")

# Conditional expression (ternary operator)
result = "Even" if x % 2 == 0 else "Odd"
print(result)  # Even
```

Loops

For Loops:

```python
# Iterating over a sequence
for i in range(5):
    print(i)  # Prints 0, 1, 2, 3, 4

# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Enumerate for index and value
for index, value in enumerate(fruits):
    print(f"Index {index}: {value}")
```

While Loops:

```python
# Basic while loop
count = 0
while count < 5:
    print(count)
    count += 1

# Break and continue
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 3:
        continue  # Skip 3
    if num == 5:
        break     # Stop at 5
    print(num)    # Prints 1, 2, 4
```

4. Functions and Modules

Function Definition and Use

```python
# Basic function
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))  # Hello, Alice!

# Default parameters
def greet_with_title(name, title="Mr."):
    return f"Hello, {title} {name}!"

print(greet_with_title("Smith"))         # Hello, Mr. Smith!
print(greet_with_title("Johnson", "Dr.")) # Hello, Dr. Johnson!

# Variable arguments
def sum_all(*args):
    return sum(args)

print(sum_all(1, 2, 3, 4))  # 10

# Keyword arguments
def person_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

person_info(name="Alice", age=30, country="USA")
```

Lambda Functions

```python
# Anonymous function
square = lambda x: x**2
print(square(5))  # 25

# Used with map
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # [1, 4, 9, 16]

# Used with filter
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4]
```

Modules and Imports

```python
# Importing a module
import math
print(math.sqrt(16))  # 4.0

# Importing specific functions
from math import pi, sin
print(pi)         # 3.141592653589793
print(sin(pi/2))  # 1.0

# Aliasing
import numpy as np
arr = np.array([1, 2, 3])
print(arr)  # [1 2 3]
```

5. Data Structures

Lists

```python
# Creating lists
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]

# Accessing elements
print(fruits[0])    # apple
print(fruits[-1])   # cherry
print(fruits[1:3])  # ['banana', 'cherry']

# List methods
fruits.append("orange")  # Add to end
fruits.insert(1, "mango")  # Insert at position
fruits.remove("banana")    # Remove by value
popped = fruits.pop()      # Remove and return last item
fruits.sort()              # Sort in place

# List comprehensions
squares = [x**2 for x in range(10)]
even_squares = [x**2 for x in range(10) if x % 2 == 0]
```

Tuples

```python
# Creating tuples (immutable)
point = (10, 20)
rgb = (255, 0, 0)

# Accessing elements
print(point[0])  # 10

# Tuple unpacking
x, y = point
r, g, b = rgb

# Single item tuple
singleton = (42,)  # Note the comma
```

Dictionaries

```python
# Creating dictionaries
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

# Accessing values
print(person["name"])  # Alice
print(person.get("age"))  # 30
print(person.get("country", "Unknown"))  # Unknown (default)

# Dictionary methods
keys = person.keys()
values = person.values()
items = person.items()  # Returns (key, value) pairs

# Adding/updating items
person["email"] = "alice@example.com"
person["age"] = 31

# Dictionary comprehension
squares_dict = {x: x**2 for x in range(5)}
```

Sets

```python
# Creating sets
fruits = {"apple", "banana", "cherry"}
numbers = {1, 2, 3, 4, 5}

# Set operations
fruits.add("orange")
fruits.remove("banana")
print("apple" in fruits)  # True

# Set operations
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)  # Union: {1, 2, 3, 4, 5}
print(a & b)  # Intersection: {3}
print(a - b)  # Difference: {1, 2}
```

6. File Handling

Reading Files

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

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

Writing Files

```
# Writing to a file
with open('output.txt', 'w') as file:
    file.write("Hello, world!\n")
    file.write("This is a test file.")

# Appending to a file
with open('output.txt', 'a') as file:
    file.write("\nAppending more text.")
```

Working with CSV Files

```
import csv

# Reading CSV
with open('data.csv', 'r') as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)

# Writing CSV
data = [
    ['Name', 'Age', 'Country'],
    ['Alice', 30, 'USA'],
    ['Bob', 25, 'Canada']
]
with open('output.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerows(data)
```

7. Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm built around the concept of "objects" that contain data and code.

Classes and Objects

```python
# Defining a class
class Person:
    # Class attribute
    species = "Homo sapiens"

    # Constructor method
    def __init__(self, name, age):
        # Instance attributes
        self.name = name
        self.age = age

    # Instance method
    def introduce(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old."

    # Instance method with parameters
    def celebrate_birthday(self):
        self.age += 1
        return f"Happy Birthday! Now {self.name} is {self.age} years old."

# Creating objects (instances)
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

# Accessing attributes
print(person1.name)  # Alice
print(person2.age)   # 25
print(Person.species)  # Homo sapiens

# Calling methods
print(person1.introduce())  # Hi, I'm Alice and I'm 30 years old.
print(person2.celebrate_birthday())  # Happy Birthday! Now Bob is 26 years old.
```

Inheritance

```python
# Parent class
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        print("Some generic animal sound")

# Child class
class Dog(Animal):
    def __init__(self, name, breed):
        # Call parent constructor
        super().__init__("Canine")
        self.name = name
        self.breed = breed

    # Override parent method
    def make_sound(self):
        print("Woof!")

    # New method
    def fetch(self):
        print(f"{self.name} is fetching the ball!")

# Create instance
my_dog = Dog("Rex", "Golden Retriever")
print(my_dog.species)  # Canine
my_dog.make_sound()    # Woof!
my_dog.fetch()         # Rex is fetching the ball!
```

Encapsulation

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self._balance = balance  # Protected attribute

    # Getter method
    def get_balance(self):
        return self._balance

    # Setter method
    def set_balance(self, amount):
        if amount >= 0:
            self._balance = amount
        else:
            raise ValueError("Balance cannot be negative")

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            return f"Deposited ${amount}. New balance: ${self._balance}"
        else:
            raise ValueError("Deposit amount must be positive")

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            return f"Withdrew ${amount}. New balance: ${self._balance}"
        else:
            raise ValueError("Invalid withdrawal amount")

# Using the BankAccount class
account = BankAccount("Alice", 1000)
print(account.get_balance())  # 1000
account.deposit(500)          # Deposited $500. New balance: $1500
account.withdraw(200)         # Withdrew $200. New balance: $1300
```

Polymorphism

```python
class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius

# Polymorphic function
def print_shape_info(shape):
    print(f"Area: {shape.area()}")
    print(f"Perimeter: {shape.perimeter()}")

# Using polymorphism
rect = Rectangle(5, 10)
circ = Circle(7)

print_shape_info(rect)  # Works for Rectangle
print_shape_info(circ)  # Works for Circle
```

8. Exception Handling

Exception handling allows you to gracefully deal with errors in your code.

Basic Exception Handling

```
# Try-except block
try:
    x = 10 / 0  # This will cause a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")

# Try-except-else-finally
try:
    number = int(input("Enter a number: "))
    result = 100 / number
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    # Runs if no exceptions were raised
    print(f"100 divided by {number} is {result}")
finally:
    # Always runs, regardless of whether an exception occurred
    print("This code always executes")
```

Handling Multiple Exceptions

```
try:
    # Code that might raise different exceptions
    file = open("nonexistent_file.txt", "r")
    content = file.read()
    number = int(content.strip())
    result = 10 / number
except FileNotFoundError:
    print("The file doesn't exist!")
except ValueError:
    print("The file doesn't contain a valid number!")
except ZeroDivisionError:
    print("The number in the file is zero, and we can't divide by it!")
except Exception as e:
    # Catch any other exception
    print(f"An unexpected error occurred: {e}")
```

Raising Exceptions

```python
def validate_age(age):
    if not isinstance(age, int):
        raise TypeError("Age must be an integer")
    if age < 0:
        raise ValueError("Age cannot be negative")
    if age > 150:
        raise ValueError("Age seems too high")
    return True

# Using the function with exception handling
try:
    validate_age("twenty")
except TypeError as e:
    print(e)  # Age must be an integer

try:
    validate_age(-5)
except ValueError as e:
    print(e)  # Age cannot be negative
```

Creating Custom Exceptions

```python
# Custom exception class
class InsufficientFundsError(Exception):
    """Raised when a bank account has insufficient funds for a withdrawal"""
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        self.deficit = amount - balance
        super().__init__(f"Insufficient funds: balance=${balance}, attempted to with

# Using the custom exception
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsError(self.balance, amount)
        self.balance -= amount
        return self.balance

# Example
account = BankAccount(50)
try:
    account.withdraw(100)
except InsufficientFundsError as e:
    print(e)  # Insufficient funds: balance=$50, attempted to withdraw $100, deficit
    # Handle the error (perhaps offer overdraft)
```

9. Working with APIs

APIs (Application Programming Interfaces) allow your Python programs to interact with web services and external data sources.

Making HTTP Requests

```python
import requests

# Basic GET request
response = requests.get("https://api.example.com/data")
print(response.status_code)  # 200 if successful
print(response.text)         # Raw response text
print(response.json())       # Parse JSON response

# GET request with parameters
params = {
    "query": "python",
    "limit": 10
}
response = requests.get("https://api.example.com/search", params=params)
print(response.url)  # URL with parameters added

# POST request with JSON data
data = {
    "name": "John Doe",
    "email": "john@example.com"
}
response = requests.post("https://api.example.com/users", json=data)
print(response.json())  # Server response
```

Handling API Authentication

```python
# API with API Key
api_key = "your_api_key_here"
headers = {
    "Authorization": f"Bearer {api_key}"
}
response = requests.get("https://api.example.com/protected", headers=headers)

# Basic authentication
from requests.auth import HTTPBasicAuth
response = requests.get(
    "https://api.example.com/secure",
    auth=HTTPBasicAuth("username", "password")
)

# OAuth2 (simplified)
token_response = requests.post(
    "https://api.example.com/oauth/token",
    data={
        "grant_type": "client_credentials",
        "client_id": "your_client_id",
        "client_secret": "your_client_secret"
    }
)
token = token_response.json()["access_token"]
headers = {"Authorization": f"Bearer {token}"}
response = requests.get("https://api.example.com/resource", headers=headers)
```

Working with JSON

```python
import json

# Parsing JSON
json_string = '{"name": "John", "age": 30, "city": "New York"}'
data = json.loads(json_string)
print(data["name"])  # John

# Creating JSON
user = {
    "name": "Alice",
    "age": 25,
    "is_student": False,
    "courses": ["Python", "Data Science"],
    "address": {
        "city": "Boston",
        "zipcode": "02108"
    }
}
json_string = json.dumps(user, indent=2)
print(json_string)

# Writing JSON to a file
with open("user.json", "w") as file:
    json.dump(user, file, indent=2)

# Reading JSON from a file
with open("user.json", "r") as file:
    loaded_user = json.load(file)
```

Asynchronous API Requests

```python
import asyncio
import aiohttp

async def fetch_data(session, url):
    async with session.get(url) as response:
        return await response.json()

async def main():
    # List of APIs to call
    urls = [
        "https://api.example.com/users",
        "https://api.example.com/products",
        "https://api.example.com/orders"
    ]

    async with aiohttp.ClientSession() as session:
        # Create tasks for all URLs
        tasks = [fetch_data(session, url) for url in urls]

        # Wait for all tasks to complete
        results = await asyncio.gather(*tasks)

        # Process results
        for i, result in enumerate(results):
            print(f"Result from {urls[i]}: {result}")

# Run the async main function
if __name__ == "__main__":
    asyncio.run(main())
```