IMAGE and VIDEO PROCCESSING

(Introduction to Python)

Asst. Prof. TUĞÇEM PARTAL

Computer Engineering

Recep Tayyip Erdoğan University

March 6, 2025

Overview

1. Introduction to Python
2. Python IDEs
3. Objects
4. Expressions
5. Operators
6. Console Usage
7. math Library
8. NumPy Library
9. Data Types

What is Python?

- **High-level, interpreted language:** Code is interpreted directly without compiling to machine language.
- **Readable and simple syntax:** Indentation and straightforward structures make the code more understandable.
- **Extensive library support:** Useful in data science, web development, automation, artificial intelligence, and more.
- **Rapid prototyping:** Allows quick solutions with minimal code.

History and Philosophy of Python

- **Creator:** Developed by Guido van Rossum.
- **First Release:** Published in 1991.
- **Philosophy:** Guided by the principle *"There should be one– and preferably only one –obvious way to do it"*, prioritizing readability and simplicity.
- **Community:** Open-source nature with a large user and developer community.

Companies and Fields Where Python Is Used

Companies Using Python:

- **Google:** Infrastructure, data processing, search engine services.
- **YouTube:** Video processing, content management, analytics.
- **Facebook/Meta:** Data analysis, automation, AI applications.
- **Instagram:** Backend development, web services.

Fields of Use:

- **Web Development:** Frameworks like Django, Flask.
- **Data Science and Machine Learning:** NumPy, Pandas, scikit-learn, TensorFlow.
- **Artificial Intelligence:** Deep learning, natural language processing.
- **Automation:** Scripting, test automation, system administration.

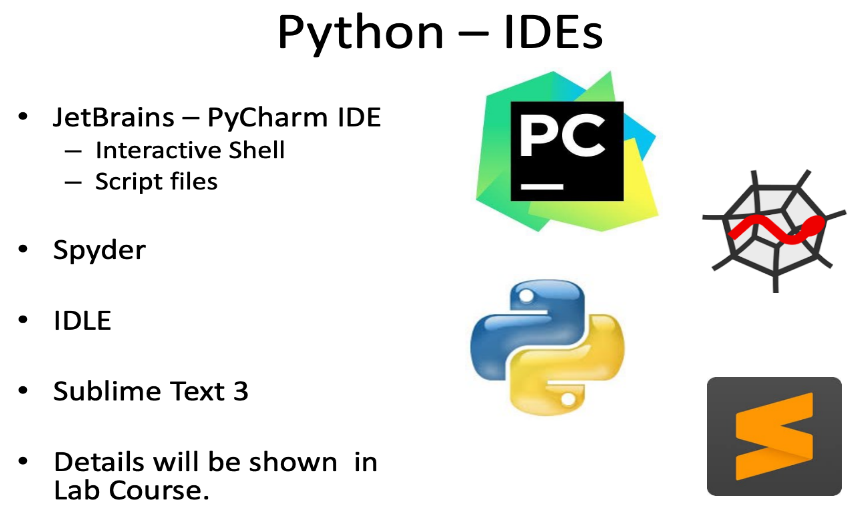
Companies and Fields Where Python Is Used

Companies Using Python:

- **Spotify:** Data analysis, recommendation systems, music streaming.
- **Netflix:** Content recommendation engines, data analytics.
- **Amazon:** Web services, automation, cloud infrastructure.

Fields of Use:

- **Scientific Computing:** Simulation, modeling, high-performance computing.
- **Finance and Economics:** Algorithmic trading, data analysis.
- **Education:** Teaching programming, research projects.



Python – IDEs

- JetBrains – PyCharm IDE
  - Interactive Shell
  - Script files
- Spyder
- IDLE
- Sublime Text 3
- Details will be shown in Lab Course.

Basic Editors (1/2)

- **Thonny**
  - Very simple and user-friendly for beginners.
  - Automatic debugging support, simple interface.
  - **Download:** `https://thonny.org/`
- **IDLE**
  - Python's standard, minimal editor.
  - Comes with Python installation.
  - **Download:** `https://www.python.org/` (IDLE is included when you install Python)
- **Jupyter Notebook**
  - Interactive code execution with text and graphics in one place.
  - Used for data science, analysis, and education.
  - **Download:** `https://jupyter.org/`

Advanced Editors (2/2)

- **Visual Studio Code (VS Code)**
  - Modern interface with rich plugin and theme support.
  - Integrated terminal, Git support; ideal for small to medium projects.
  - **Download:** `https://code.visualstudio.com/`
- **PyCharm (Community & Professional)**
  - Community Edition: Free, basic Python development features.
  - Professional Edition: Additional features (web development, database integration, etc.); paid.
  - **Download:** `https://www.jetbrains.com/pycharm/download/`
- **Eclipse + PyDev**
  - Multi-language support and highly extensible.
  - Preferred for large and complex projects.
  - **Download Eclipse:** `https://www.eclipse.org/`
    Download PyDev: `https://www.pydev.org/`

Python Programs

- A **program** is a sequence of definitions and commands
  - Definitions are *evaluated*.
  - Commands are *executed* by the Python interpreter in a shell.
- **Commands (statements)** instruct the interpreter to do something.
- They can be typed directly in a shell or stored in a file that is read into the shell and evaluated.
  - Problem Set 0 will introduce you to these in Anaconda.

Objects(1/2)

- Programs manipulate **data objects**.
- Objects have a **type** that defines the kinds of things programs can do to them.
  - `3.0`
    - Is a number.
    - We can add/subtract/multiply/divide/exponentiate, etc.
  - `'Ana'`
    - Is a sequence of characters (a **string**).
    - We can grab substrings, but we *can't* divide it by a number.

Objects (2/2)

- **Scalar** (cannot be subdivided)
  - Numbers: `8.3`, `2`
  - Truth values: `True`, `False`
- **Non-scalar** (have internal structure that can be accessed)
  - Lists
  - Dictionaries
  - Sequence of characters: `"abc"`

Scalar Objects

- `int` – represent **integers**, e.g., `5`, `-100`
- `float` – represent **real numbers**, e.g., `3.27`, `2.0`
- `bool` – represent **Boolean values** `True` and `False`
- `NoneType` – special and has one value, `None`
- Can use `type()` to see the type of an object:

Examples

```
>>> type(5)
int
>>> type(3.0)
float
```

YOU TRY IT!

- In your console, find the type of:
  - `1234`
  - `8.99`
  - `9.0`
  - `True`
  - `False`

TYPE CONVERSIONS (CASTING)

- Can **convert object of one type to another**
  - `float(3)` casts the int 3 to float 3.0
  - `int(3.9)` casts (note the truncation!) the float 3.9 to int 3
  - Some operations perform **implicit** casts
    - `round(3.9)` returns the int 4

YOU TRY IT!

- In your console, find the type of:
  - `float(123)`
  - `round(7.9)`
  - `float(round(7.2))`
  - `int(7.2)`
  - `int(7.9)`

EXPRESSIONS

- **Combine objects and operators** to form expressions
  - 3+2
  - 5/3
- An expression has a **value**, which has a **type**
  - 3+2 has value `5` and type `int`
  - 5/3 has value `1.666667` and type `float`
- Python **evaluates expressions** and stores the value. It *doesn't* store expressions!
- Syntax for a simple expression:
  `<object> <operator> <object>`

OPERATORS

OPERATORS on `int` and `float`

- `i+j` → the **sum**
- `i-j` → the **difference**
- `i*j` → the **product**

if both are ints, result is int
if either or both are floats, result is float

- `i/j` → **division**

result is always a float

- `i//j` → **floor division**

What is type of output?

- `i%j` → the **remainder** when `i` is divided by `j`

Python Arithmetic and Logical Operators

Command	Output
»> 7 // 3	2
»> 1 / 2	0.5
»> 2 ** 3	8
»> (1+2)*3-4/2	7
»> import math »> math.sqrt(16)	4.0
»> x = 10 »> x -= 3 »> print(x)	7

Python Arithmetic and Logical Operators

»> 5 <= 7	True
»> 5 == 7	False
»> 5 >= 7	False
»> (5 < 7) and (7 < 10)	True
»> (5 < 7) or (7 > 10)	True
»> not(5 < 7)	False

Python Console Examples: String Operations

Command	Output
»> s = "Hello World" »> s.split()	[’Hello’, ’World’]
»> s.upper()	"HELLO WORLD"
»> s.lower()	"hello world"
»> len(s)	11
»> type(s) »> type(123) »> type(3.14)	<class ’str’> <class ’int’> <class ’float’>

Python Console Examples: List Operations

Command	**Output**

`»> lst = [1, 2, 3, 4, 5]`
`»> lst.append(6)`
`»> lst`
	`[1, 2, 3, 4, 5, 6]`

`»> lst[2]`	`3`
`»> lst[1:4]`	`[2, 3, 4]`

Python math Library

- **Description:**
  Python's math library is used to perform fundamental mathematical operations. Its name is derived from the word "mathematics," and it includes functions such as square root, trigonometric functions, logarithms, factorial, and more. It also provides various constants and rounding functions.
- **Notable Functions:** math.sqrt, math.sin, math.log, math.factorial, math.floor, math.ceil, math.degrees, math.hypot, math.comb, math.gcd

Python math Library

Command	Output
»> import math »> math.sqrt(25)	5.0
»> math.sin(math.pi/2)	1.0
»> math.log(100, 10)	2.0
»> math.factorial(5)	120
»> math.floor(3.7) »> math.ceil(3.1)	3, 4
»> math.degrees(math.pi)	180.0
»> math.hypot(3,4)	5.0

Python NumPy Library

- **Description:**
  NumPy (Numerical Python) allows high-performance mathematical operations on large, multi-dimensional arrays. It is widely used in scientific computing, data analysis, and machine learning projects. Thanks to its C-based infrastructure, NumPy runs at high speed and supports vectorized operations.
- **Notable Functions:** numpy.array, numpy.arange, numpy.linspace, numpy.ones, numpy.zeros, numpy.eye, numpy.dot, numpy.sum, numpy.mean, numpy.std, numpy.reshape, numpy.transpose, numpy.sqrt, numpy.min, numpy.max, numpy.sort, numpy.concatenate, numpy.random

Python NumPy Library

Command	Output
»> import numpy as np »> a = np.array([1, 2, 3]) »> a	array([1, 2, 3])
»> np.arange(0, 10, 2)	array([0, 2, 4, 6, 8])
»> np.linspace(0, 1, 5)	array([0. , 0.25, 0.5 , 0.75, 1.
»> b = np.array([[1,2],[3,4]]) »> np.transpose(b)	array([[1, 3], [2, 4]])
»> np.sqrt(a)	array([1. , 1.41421356, 1.7320508

Multiple Assignment Examples in Python

- **Parallel Assignment**

  ```
  x, y = 2, 3
  print("x =", x, "y =", y)
  ```

  This method assigns values to two variables in a single line instead of separate assignments as in C.

- **Chain Assignment:**

  ```
  x = y = z = 1
  print("x =", x, "y =", y, "z =", z)
  ```

  This assignment method sets the same value to all variables at once.

- **Unpacking Values:**

  ```
  a, b, c = [10, 20, 30]
  print("a =", a, "b =", b, "c =", c)
  ```

  The values inside a list or tuple are assigned sequentially to the variables.

Multiple Assignment Examples in Python

- **Variable Swapping:**

```
x, y = 5, 10
print("Before: x =", x, "y =", y)
x, y = y, x
print("After: x =", x, "y =", y)
```

  This method swaps variable values without using an extra temporary variable.

- **Extended Unpacking:**

```
a, *b, c = (1, 2, 3, 4, 5)
print("a =", a)
print("b =", b)
print("c =", c)
```

  Here, a takes the first value, c takes the last value, and b gathers the remaining values as a list.

MATLAB and Python: Fibonacci Numbers Less Than 200

MATLAB Code Example:

```
a = 0;
b = 1;
c = 0;

while c < 200
c = a + b;
fprintf(’%d ’, c);
a = b;
b = c;
end

fprintf(’\n’);
```

Python Code Example:

```
a, b = 0, 1
while b < 200:
print(b)
a, b = b, a + b
```

Data Types

Variable Name	**Type**	**Description and Features**
number	int	Represents integer values. (Homogeneous)
ratio	float	Represents floating-point numbers. (Homogeneous)
name	str	Used to store textual data. (Homogeneous)
flag	bool	Represents a Boolean value. (Homogeneous)
list_var	list	Ordered data structure. **Feature:** Can store values of different types (heterogeneous).

Data Types Cont.

tuple_var	tuple	Immutable ordered data structure. **Feature:** Can contain values of different types (heterogeneous).
dictionary	dict	Stores key-value pairs. **Feature:** Both keys and values can be of different types.
set_var	set	Data structure composed of unique elements. **Feature:** Can contain elements of different types (heterogeneous), although homogeneous data is typically preferred.

Data Types: Examples

```
# Homogeneous data examples:
number    = 5             # int
ratio     = 3.14          # float
name      = "Ahmet"       # str
flag      = True          # bool
```

Heterogeneous data examples:

```
#list: can store different types
  list_var  = [1, "two", 3.0, False]
#tuple:can store different types
  tuple_var = (1, "two", 3.0, False)
#dictionary: keys and values can be of different types
   dictionary = {"number": 5, "name": "Ahmet"}
#set:can contain elements of different types
   set_var   = {1, "two", 3.0}
```

Data Types: Examples

```
# To check data types:
print(type(number))      # <class ’int’>
print(type(ratio))       # <class ’float’>
print(type(flag))        # <class ’bool’>
print(type(list_var))    # <class ’list’>
print(type(dictionary)) # <class ’dict’>
```

Tuple

- **Definition:** A tuple is an ordered and **immutable** data structure. It is defined using parentheses `()`.
- **Properties:**
  - Once created, its contents cannot be changed.
  - It is fast and ideal for storing constant data.
- **Example:** `t = (1, "two", 3.0)`
- **Built-in Methods:**
  - `t.count(x)`: Returns the number of times `x` appears in the tuple.
  - `t.index(x)`: Returns the index of the first occurrence of `x` in the tuple.

Tuple Examples

```
# Example 1: Simple Tuple
t = (1, 2, 3, 2, 4)
print(t)
# Output: (1, 2, 3, 2, 4)

# Example 2: count() and index() methods
print(t.count(2))
# Output: 2
print(t.index(3))
# Output: 2
```

Tuple Examples Cont.

```
# Example 3: Tuple Unpacking
a, b, c, d, e = t
print(a, b, c, d, e)
# Output: 1 2 3 2 4

# Example 4: Nested Tuple
nested = (1, (2, 3), (4, (5, 6)))
print(nested)
# Output: (1, (2, 3), (4, (5, 6)))
```

Nested Tuple Example A tuple can contain other tuples; this is called a "nested tuple". For example:

```
nested = (1, (2, 3), (4, (5, 6)))
print("Complete tuple:", nested)
# Output: Complete tuple: (1, (2, 3), (4, (5, 6)))
```

In this example:

- `nested[0]` is 1.
- `nested[1]` is a tuple: `(2, 3)`.
- `nested[2]` is a tuple: `(4, (5, 6))`.

Nested Tuple Example Cont.

Accessing elements in nested tuples:

```
# Second element (tuple): (2, 3)
print("Second element:", nested[1])
# Second element of the third tuple: (5, 6)
print("Second element of the third tuple:", nested[2][1])
# The innermost element: 5
print("Innermost element:", nested[2][1][0])
```

This method lets you create multi-layered data structures with tuples and access any element using indices.

List

- **Definition:** A list is an ordered and **mutable** data structure. It is defined using square brackets `[]`.
- **Properties:**
  - It can store heterogeneous data (different data types).
  - Suitable for dynamic operations such as adding, removing, and updating elements.
- **Example:** `lst = [1, "two", 3.0, True]`

List Cont.

- **Special Methods:**
  - `lst.append(x)`: Appends `x` to the end of the list.
  - `lst.extend([x, y, ...])`: Extends the list with the given list.
  - `lst.insert(i, x)`: Inserts `x` at the specified index `i`.
  - `lst.pop([i])`: Removes and returns the element at index `i`; if no index is provided, removes the last element.
  - `lst.remove(x)`: Removes the first occurrence of `x` from the list.
  - `lst.copy()` or `lst[:]`: Creates a copy of the list.
  - `lst.clear()`: Removes all elements from the list.

List Examples (1/2)

```
# Creating a simple list
lst = [1, 2, 3, 4]
print(lst)
# Output: [1, 2, 3, 4]

# Appending an element
lst.append(5)
print(lst)
# Output: [1, 2, 3, 4, 5]
```

List Examples (1/2) Cont.

```
# Extending the list
lst.extend([6, 7])
print(lst)
# Output: [1, 2, 3, 4, 5, 6, 7]

# Inserting an element at a specific index
lst.insert(0, 0)
print(lst)
# Output: [0, 1, 2, 3, 4, 5, 6, 7]
```

List Examples (2/2)

```
# Removing an element using pop() (removes the last element)
lst.pop()
print(lst)
# Output: [0, 1, 2, 3, 4, 5, 6]

# Removing an element using remove() (removes the first occurrence)
lst.remove(3)
print(lst)
# Output: [0, 1, 2, 4, 5, 6]
```

List Examples (2/2)

```
# Copying a list
lst_copy = lst.copy()
print(lst_copy)
# Output: [0, 1, 2, 4, 5, 6]

# Clearing the list
lst.clear()
print(lst)
# Output: []
```

Set

- **Definition:** A set is an unordered data structure composed of unique elements. It can be created using `{}` or `set()`.
- **Properties:**
  - Elements are unique; duplicate values are automatically removed.
  - Being unordered, sets cannot be indexed.
- **Example:** `s = {1, 2, 2, 3, "four"}` (Result: `{1, 2, 3, "four"}`)
- **Special Methods:**
  - `s.add(x)`: Adds `x` to the set.
  - `s.update({x, y, ...})`: Adds the given elements to the set.
  - `s.remove(x)`: Removes `x` from the set; raises an error if `x` is not found.
  - `s.discard(x)`: Removes `x` without raising an error if `x` is not found.
  - `s.pop()`: Removes and returns a random element from the set.
  - `s.clear()`: Clears all elements from the set.
  - `s.copy()`: Creates a copy of the set.

Set Examples (1/2)

```
# Duplicate elements are automatically removed in a set.
s = {1, 2, 2, 3, 4}
print("Set:", s)
# Example output: Set: {1, 2, 3, 4}

# Adding an element using add()
s.add(5)
print("After add(5):", s)

# Adding multiple elements using update()
s.update({6, 7})
print("After update({6,7}):", s)
```

Set Examples (1/2)

```
# Removing a specific element using remove()
s.remove(3)
print("After removal:", s)

# Removing an element without error using discard()
s.discard(10)
print("After discard:", s)
```

Set Examples (2/2)

```
# Removing a random element using pop()
elem = s.pop()
print("Element removed by pop():", elem)
print("After pop():", s)

# Copying a set using copy()
s_copy = s.copy()
print("Copy:", s_copy)

# Clearing the set using clear()
s.clear()
print("After clearing:", s)  #Example output: After clearing: set()
```

Dictionary

- **Definition:** A dictionary is a mutable and (in Python 3.7 and later) ordered data structure that stores key-value pairs. It is defined using `{}`.
- **Properties:**
  - Keys are unique and typically immutable (e.g., str, int, tuple).
  - Values can be of any type.
- **Example:** `d = {"name": "Ahmet", "age": 30, "status": True}`

Dictionary Cont.

- **Special Methods:**
  - `d.keys()` : Returns all keys.
  - `d.values()` : Returns all values.
  - `d.items()` : Returns key-value pairs.
  - `d.get(key)` : Returns the value for the specified key.
  - `d.update({...})` : Updates the dictionary.
  - `d.pop(key)` : Removes the specified key and its value.
  - `d.popitem()` : Removes and returns a key-value pair (usually the last one).
  - `d.clear()` : Clears all items from the dictionary.
  - `d.copy()` : Creates a copy of the dictionary.

Dictionary Examples (1/3)

```
# Creating a dictionary
d = {"name": "Ahmet", "age": 30, "city": "Ankara"}
print("Dictionary:", d)
# Output: Dictionary: {’name’: ’Ahmet’, ’age’: 30, ’city’: ’Ankara’}

# Getting keys, values, and items
print("Keys:", d.keys())
print("Values:", d.values())
print("Items:", d.items())
# Output:
# Keys: dict_keys([’name’, ’age’, ’city’])
# Values: dict_values([’Ahmet’, 30, ’Ankara’])
# Items: dict_items([(’name’, ’Ahmet’), (’age’, 30), (’city’, ’Ankar
```

Dictionary Examples (2/3)

```
# Updating the dictionary
d.update({"age": 35, "profession": "Engineer"})
print("Updated dictionary:", d)
# Output: Updated dictionary: {’name’: ’Ahmet’, ’age’: 35, ’city’: ’

# Retrieving a value with get()
print("name:", d.get("name"))
# Output: name: Ahmet

# Removing a key-value pair using pop()
age = d.pop("age")
print("Removed age:", age)
print("Dictionary:", d)
# Output: Removed age: 35
# Dictionary: {’name’: ’Ahmet’, ’city’: ’Ankara’, ’profession’: ’Eng
```

Dictionary Examples (3/3)

```
# Copying and clearing the dictionary
d_copy = d.copy()
print("Copy:", d_copy)
d.clear()
print("Cleared dictionary:", d)
# Output: Cleared dictionary: {}
```

When to Use Which Data Structure?

- **List:**
  - When you need to store ordered data and perform indexing.
  - When dynamic operations like adding, removing, or updating elements are required.
  - **Example Use:** A list of student names or sequential data processing.
- **Tuple:**
  - When the data should remain constant and unchangeable.
  - Useful for returning multiple values from functions or storing fixed configuration data.
  - **Example Use:** (x, y) coordinates or fixed setting parameters.

When to Use Which Data Structure?

- **Set:**
    - When you need to store unique (non-repeating) elements.
    - When performing mathematical set operations like union, intersection, or difference.
    - **Example Use:** Unique user IDs or tags.
- **Dictionary:**
    - When you need fast access to data via key-value pairs.
    - For flexible data modeling and configuration settings.
    - **Example Use:** User information, phone directories, configuration parameters.

Below is an example demonstrating how to convert a **tuple** to a **list** and a **list** to a **tuple**.

```
# Converting a tuple to a list
t = (1, 2, 3)
lst = list(t)
print("List:", lst)
# Output: List: [1, 2, 3]

# Converting a list to a tuple
lst2 = [4, 5, 6]
t2 = tuple(lst2)
print("Tuple:", t2)
# Output: Tuple: (4, 5, 6)
```

Example This example shows conversions between list, set, and dictionary:

```
# Converting a list to a set:
# Duplicate values are automatically removed.
lst = [1, 2, 2, 3, 3, 3]
s = set(lst)
print("Set:", s)
# Output: Set: {1, 2, 3}
```

Example Cont.

```
# Converting a list to a dictionary:
#The list must consist of (key, value) pairs.
lst2 = [("a", 1), ("b", 2)]
d = dict(lst2)
print("Dictionary:", d)
# Output: Dictionary: {'a': 1, 'b': 2}

# Converting dictionary keys to a list:
keys = list(d.keys())
print("Keys:", keys)
# Output: Keys: ['a', 'b']
```