Lab 01
BDA

Python Crash Course

Variables

*In Python, a **variable** is a named reference to a value stored in memory. Variables allow you to store, modify, and retrieve data within a program*

Variables

Properties of Variables

They do not require explicit declaration (dynamic typing).

They can hold different types of data (integers, floats, strings, lists, etc.).

The type of a variable is determined automatically based on the assigned value.

Code of example

```
x = 10 # Integer
y = 3.14 # Float
name = "Python" # String
is_active = True # Boolean
```

Variable Naming

Variable Naming Rules

Must start with a letter (a-z, A-Z) or an underscore _.

Cannot start with a number.

Can contain letters, numbers, and underscores.

Case-sensitive (myVar and myvar are different).

Cannot use Python reserved keywords (e.g., if, for, while).

Code of Example

```
a = 5
b = 10
sum_value = a + b
print(sum_value) # Output: 15
```

Type Checking and Casting

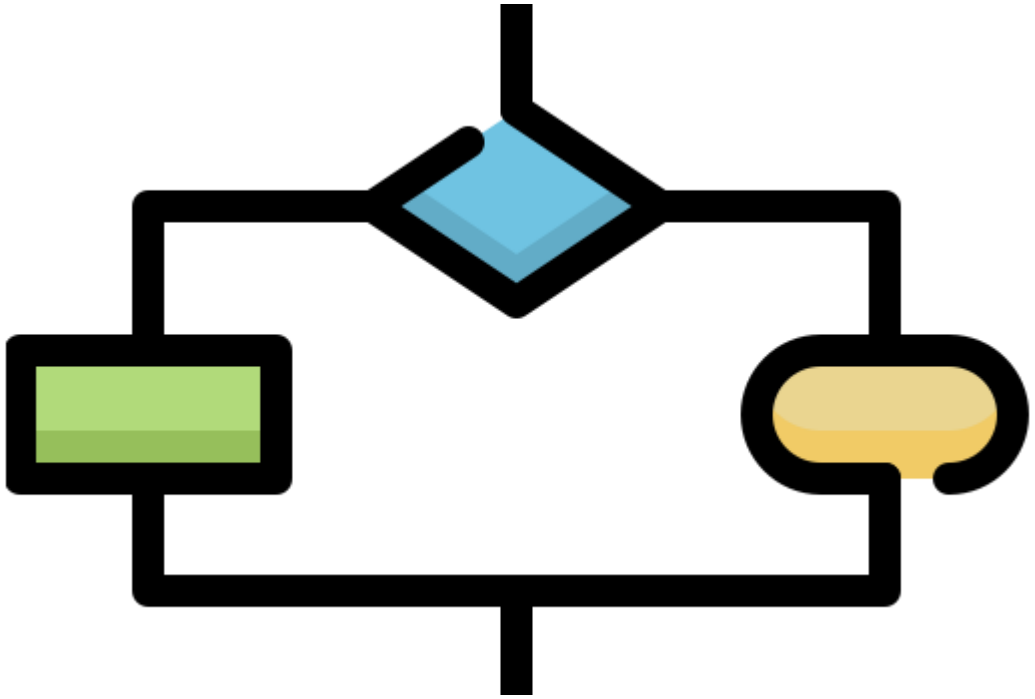
Type Checking

To check the type of a variable, use the **type()** function

Type Casting

```
x = int(3.14) # Converts float to int
print(x) # Output: 3
```

Conditional Statements



Loops allow for the repetition of code execution multiple times, either for a set number of iterations or while a condition is true.

Conditional Statements

What is conditional statements

Conditional statements allow a program to make decisions based on conditions. They evaluate expressions and execute different code blocks depending on whether the condition is **True** or **False**.

Simplest Statement

```
x = 10
if x > 5:
  print("x is greater than 5")
```

Ternary Conditional Operator

Ternary (Conditional) Operator

A short-hand way to write if-else in a single line.

Example

```
message = "x is big" if x > 5 else "x is small"
```

Nested Ifs

Nested If

An if statement inside another if statement.

Nested Ifs Example

```
if x > 5:
    if x < 20:
        print("x is between 5 and 20")
```

Loops

Loops allow for the repetition of code execution multiple times, either for a set number of iterations or while a condition is true.

Loops

Explanation

Loops allow for the repetition of code execution multiple times, either for a set number of iterations or while a condition is true. Here are the common types of loops:

Range/List/String

```
for i in range(5): # Loops from 0 to 4
  print("Iteration:", i)


fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)


for char in "Python":
     print(char)
```

While Loop

Description

Executes the loop as long as the condition is True.

Counting Down and Infinite Loop

```
x = 5
while x > 0:
  print("Countdown:", x)
  x -= 1


while True:
  user_input = input("Enter 'stop' to
exit: ")
  if user_input.lower() == "stop":
      break # Exits the loop
```

Break and Continue

Description:

break ⟶ Exits the loop immediately.

continue ⟶ Skips the current iteration and moves to the next

Break and Continue:

```
for i in range(10):
  if i == 5:
      break # Stops the loop when 5
  print(i)


for i in range(5):
    if i == 2:
        continue # Skips iteration
when i is 2
    print(i)
```

Nested Fors

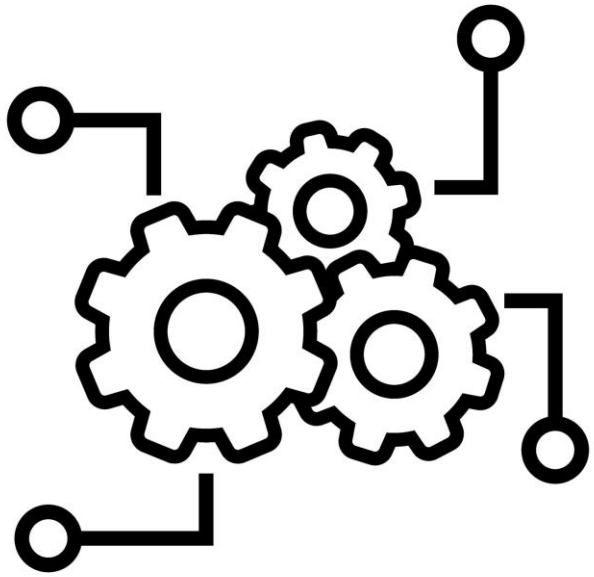
Description:

A loop inside another loop.

Example:

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

Functions



Functions are reusable blocks of code that perform a specific task. They help organize code, improve readability, and avoid repetition.

Define Function and Calling

Description:

A function is defined using the def keyword and called using its name.

Defining and Calling:

```
def greet():
    print("Hello, World!")
greet() # Calling the function
```

Functions with Parameters

Description:

Parameters allow functions to receive inputs.

Example:

```
def greet(name):
    print(f"Hello, {name}!")
    greet("Irene") # Output: Hello,
Irene!
```

Function with Return Value

Description:

A function can return a value using the return statement.

Example:

```
def square(num):
    return num * num

result = square(4)
print(result) # Output: 16
```

Function with Default Parameters

Description:

If a parameter is not provided, the default value is used.

Example:

```
def greet(name="Guest"):
    print(f"Hello, {name}!")
greet() # Output: Hello, Guest!
greet("Irene") # Output: Hello, Irene!
```

Function with Multiple Parameters

Description:

A function can take multiple parameters.

Example:

```
def add(a, b):
  return a + b
print(add(3, 7)) # Output: 10
```

Variable Length Arguments

Description:

*args → Allows passing multiple positional arguments.

**kwargs → Allows passing multiple keyword arguments.

Example:

```
def sum_all(*numbers):
    return sum(numbers)
    print(sum_all(1, 2, 3, 4, 5)) #
Output: 15


def show_info(**details):
    for key, value in details.items():
        print(f"{key}: {value}")
        show_info(name="Irene",
age=25, country="Portugal")
```

Lambda Functions

Description:

Lambda functions are small, one-line functions.

Example:

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

Recursive Functions

Description:

A function that calls itself to solve a problem.

Example:

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

Advanced Stuff

Are you ready for that?

Function Composition

Description:

Combining multiple functions to create more complex behavior.

Example:

```
def double(x):
    return x * 2

def increment(x):
    return x + 1

def compose(f, g):
    return lambda x: f(g(x))

double_then_increment =
compose(increment, double)
print(double_then_increment(3))  #
Output: 7 (3 * 2 + 1)
```

Assyncronous Functions

Description:

Asynchronous functions allow concurrent execution.

Asynchronous programming allows tasks to run **independently** without waiting for other tasks to complete. It helps improve performance in situations where tasks might take a long time, such as waiting for a network request, file I/O, or database queries.

Example:

```
import asyncio

async def say_hello():
    await asyncio.sleep(1)
    print("Hello!")

async def main():
    await asyncio.gather(say_hello(),
say_hello())

asyncio.run(main())
```

High-order Functions

Description:

Functions that operate on other functions by taking them as arguments or returning them.

Example:

```
def apply_function(func, value):
    return func(value)

print(apply_function(lambda x: x ** 2, 4))  # Output: 16
```

Generator Functions (yield)

Description:

Generators allow iteration over large datasets without storing them in memory.

Example:

```
def countdown(n):
    while n > 0:
        yield n  # Saves state and
resumes on next call
        n -= 1

gen = countdown(5)
print(next(gen))  # Output: 5
print(next(gen))  # Output: 4
```

Memory Efficient File Reading

Example:

```
def read_large_file(file_path):
    """Generator that reads a large file line by line."""
    with open(file_path, "r", encoding="utf-8") as file:
        for line in file:
            yield line.strip()  # Yield each line without storing all lines in memory

# Using the generator
file_path = "large_file.txt"  # Replace with your actual file path

for line in read_large_file(file_path):
    print(line)  # Process each line one by one
```

Memory Efficient File Writting

This example generates numbers and writes them to a file line by line.

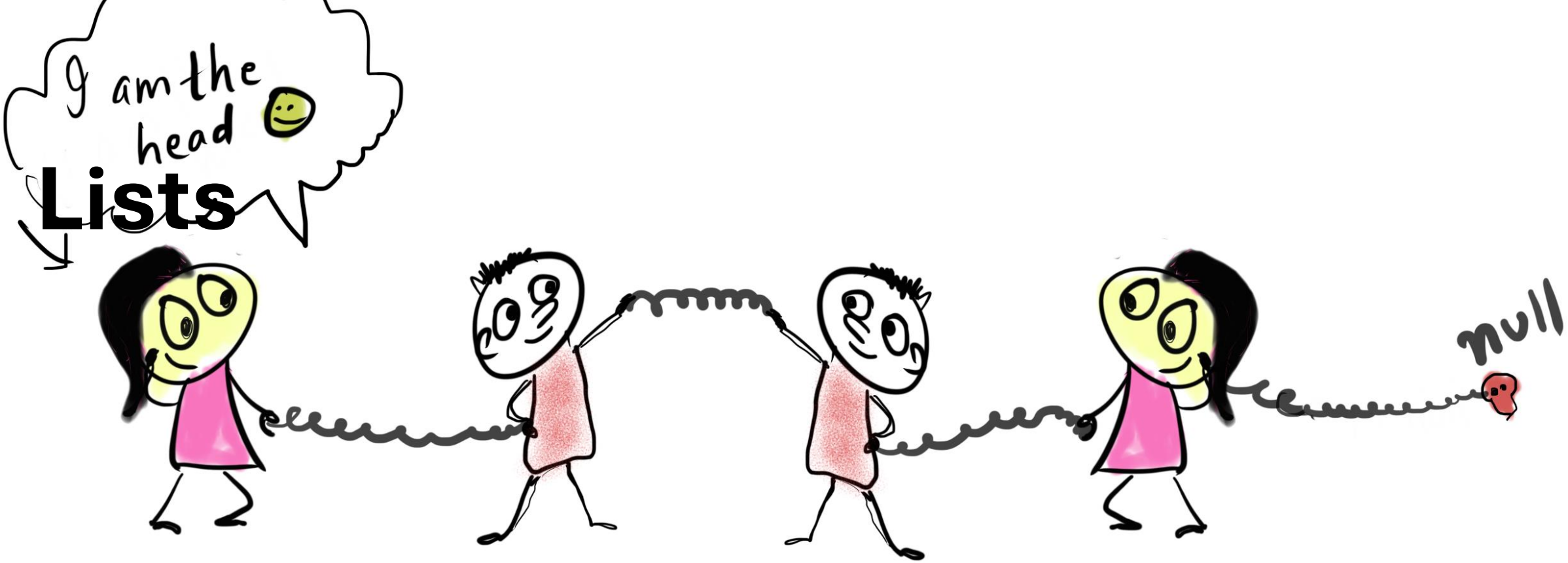
```python
def number_generator(limit):
    """Generator that yields numbers from 1 to limit."""
    for i in range(1, limit + 1):
        yield f"Number: {i}\n"

def write_to_file(file_path, generator):
    """Writes generator output to a file."""
    with open(file_path, "w", encoding="utf-8") as file:
        for line in generator:
            file.write(line)

# Using the generator
file_path = "numbers.txt"
write_to_file(file_path, number_generator(10))

print(f"Data written to {file_path}")
```

Lists



*Python **lists** are one of the most commonly used data structures, allowing you to store multiple items in a single variable.*

Creating a List

Description:

A list is created using square brackets [].

Example:

```
# Creating a list
my_list = [1, 2, 3, 4, 5]
print(my_list)


mixed_list = [1, "hello", 3.14, True]
print(mixed_list)
```

Accessing Elements

Description:

You can access elements using indexing (starting from 0).

Example:

```
my_list = ["apple", "banana", "cherry"]
print(my_list[0]) # First element
print(my_list[-1]) # Last element
```

Slicing a List

Slicing

```
numbers = [10, 20, 30, 40, 50, 60, 70]
print(numbers[1:4])    # Elements from index 1 to 3
print(numbers[:3])     # First 3 elements
print(numbers[3:])     # Elements from index 3 onwards
print(numbers[::-1])   # Reverse the list
```

You can extract parts of a list using slicing

Accessing Elements

Description:

Changing elements

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "blueberry"
print(fruits)
```

```
fruits.append("orange")         # Add at
the end
fruits.insert(1, "mango")       # Add at
index 1
fruits.extend(["grape", "kiwi"])  # Add
multiple elements
print(fruits)
```

Removing Elements

Description

remove(value) → Removes first occurrence of a value

pop(index) → Removes by index (default: last element)

del list[index] → Deletes a specific elementclear() → Removes all elements

Example

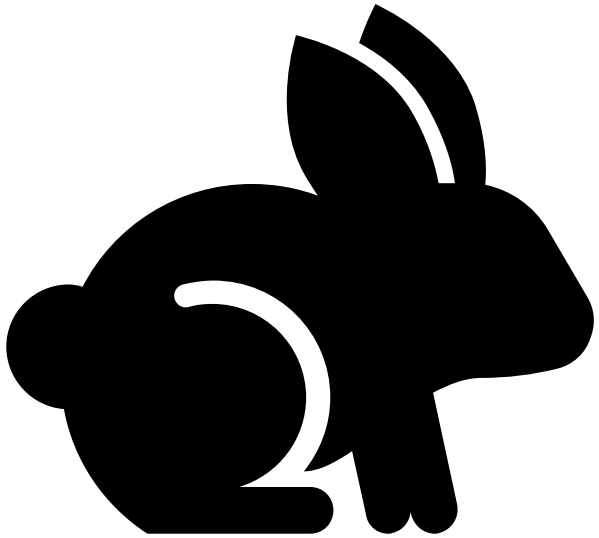
```
fruits.remove("mango")   # Remove "mango"
fruits.pop(2)            # Remove element at index 2
del fruits[0]            # Remove first element
fruits.clear()           # Empty the list
print(fruits)
```

Lists vs Dictionaries

What is faster

List (O(n)) → Searching in a list is slow because it requires scanning elements one by one (linear search).

Dictionary (O(1)) → Lookup is fast because it uses a hash table to access values instantly.



Example of Lists vs Dictionaries

Comparison

```
my_list = list(range(1, 1000000))
my_dict = {i: f"value_{i}" for i in range(1, 1000000)}

# Searching in a list (slow)
1000000 in my_list  # O(n) complexity

# Searching in a dictionary (fast)
my_dict.get(1000000)  # O(1) complexity
```

Adding Elements

Description:

List (O(1) for append,

O(n) for insert at random index)

Dictionary (O(1) for insert/update)

Example:

```
my_list.append(100)  # O(1) - Adding at the end
my_list.insert(0, 100)  # O(n) - Inserting at the beginning (shifts elements)

my_dict[100] = "new_value"  # O(1) - Insert in dictionary
```

Removing Elements

Description:

List (O(n) for removing a specific element

O(1) for pop at end)

Dictionary (O(1) for deleting a key-value pair)

Example:

```
my_list.remove(500)  # O(n) - Need to find and shift elements
my_list.pop()  # O(1) - Fast if removing last element

del my_dict[500]  # O(1) - Directly deletes key
```

Thank you

End of Lecture

Next Lecture: **Distributed Systems**